



# Spring Framework For React

Author: Pavel Šeda



# Audience

- Beginners to Spring framework, who have logical, and analytical problem-solving skills and who want to begin with learning Spring framework for the purpose of building back-end web applications



# Course Objectives

1. Foundational Spring and Jakarta/Java EE
  - a. Review IDEs (Netbeans, Eclipse/Spring Tool Suite, IntelliJ IDEA)
  - b. Review of application servers
  - c. Spring vs Jakarta/Java EE
  - d. IoC Container
  - e. Metadata configurations (Java Config, XML Config, Annotations)
  - f. Spring bean lifecycle
  - g. Spring bean scopes
  - h. Proxy objects
  - i. Reviewing of monolithic, n-tier, and microservices architectures



# Course Objectives

2. Spring and Spring Boot
  - a. Spring modules
  - b. Spring
  - c. Spring Boot
  - d. Initializing a Spring and Spring Boot application
  - e. Spring Boot bundles
  - f. Spring Boot autoconfiguration



# Course Objectives

## 3. Database Access

- a. Java Database Connectivity (JDBC)
- b. Spring JDBC Template
- c. JPA basics:
  - i. Entities, Entity relationships,
  - ii. Common annotations (@Entity, @Table, @Column, @Id, etc.)
  - iii. Fetch types (Lazy vs Eager)
- d. Spring Data
  - i. Paging and sorting
- e. Transactions



# Course Objectives

- 4. Developing Service Layer
  - a. Reviewing problem statement
  - b. Consuming another REST service (synchronously vs asynchronously)
    - i. RestTemplate
    - ii. WebResource



# Course Objectives

5. Spring Model View Controller (MVC)
  - a. Servlets
    - i. Dispatcher Servlet
  - b. Java Server Pages (JSP), Java Server Faces (JSF)
  - c. MVC Design Pattern
  - d. Common annotations:
    - i. @Controller, @ModelAttribute, @RequestMapping, @SessionAttributes
  - e. Validation (javax.validation package)
  - f. Embedded application servers (Tomcat, Jetty)



# Course Objectives

6. Spring REST Services
  - a. Rest architecture
  - b. Common annotations:
    - i. `@RestController`, `@PathVariable`, `@RequestParam`, `@RequestBody`
  - c. Data Transfer Object (DTO) classes
  - d. Jackson
  - e. Exception handling (`@RestControllerAdvice`)
  - f. Swagger Documentation
  - g. Testing Tools
    - i. Postman, SoapUI, Browser plugins





# Course Objectives

7. Spring Security
  - a. Configuration of Spring Security
  - b. Securing web requests
  - c. Storing password in Database
  - d. OAuth2 + OpenID Connect



# Introductions

Please briefly introduce yourself:

- Name
- Title or position
- Company
- Experience with Java programming and Java applications
- Reasons for attending



## **Section: Foundational Spring and Jakarta EE**



# Foundational Spring and Jakarta EE

- Review IDEs (Netbeans, Eclipse/Spring Tool Suite, IntelliJ IDEA)
- Review of application servers
- Spring vs Jakarta/Java EE
- IoC Container
- Proxy objects
- Extensible Markup Language (XML) vs Java Config
- Spring vs Spring Boot
- Reviewing of monolithic, n-tier, and microservices architectures
- Initializing a Spring application



# Why Use IDE in Java?

- Code completion
- Automatic code generation
- Syntax checking
- Great support for refactoring features
- Useful plugins (SonarLint, Database tools, ...)
- Integrated debugging
- Navigating to members by treating them as hyperlinks
- Warning as you type (i.e. some errors do not even require a compile cycle)
- Possibility to be lazy



# Integrated Development Environments (IDE)

- Netbeans
  - Created in 1996
  - Created by students project called Xelfi
  - Created in Czech Republic by students from Czech Technical University in Prague (CTU) and Charles University, leading by Roman Staněk
  - In 1999 acquired by Sun Microsystems (700 millions of dollars)
  - In 2010 Sun Microsystems was acquired by Oracle => Netbeans is Oracle IDE now
  - Advantages:
    - (+) It is free in full version
    - (+) It integrates latest trends in Java language, e.g., the modularization
  - Disadvantages
    - (-) It is slow
    - (-) It does not contain much plugins compared to other IDEs



# Integrated Development Environments (IDE)

- Eclipse
  - Created in 1998
  - Created by IBM company to “eclipse” Netbeans
  - Open Source Release in 2001
    - Fear from industry that it is too much controlled by IBM
  - In 2004 moved to newly created Eclipse Foundation to lead and develop the Eclipse community
    - Consortium of software vendors (IBM/Red Hat, Oracle, Fujitsu, SAP SE, Microsoft, ...)
  - Advantages:
    - (+) It is free in full version
    - (+) A lot of plugins included, could be used by many languages, work spaces, sweet UI
    - (+) Commonly used to develop company IDE (Spring Tool Suite, JBoss Developer Studio)
  - Disadvantages
    - (-) Sometimes plugins hell, does not feel context as IntelliJ IDEA



# Integrated Development Environments (IDE)

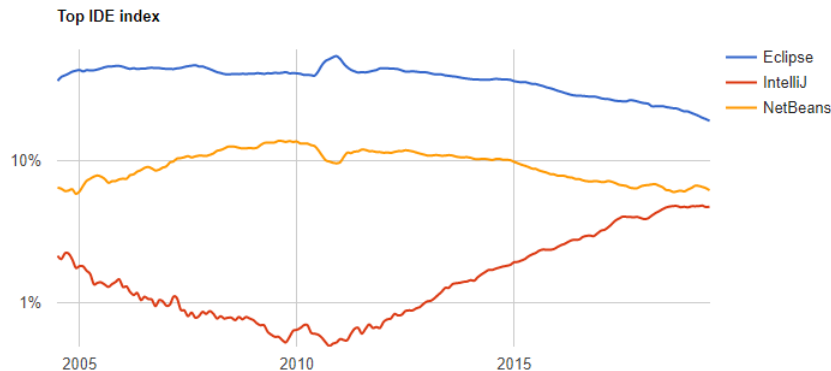
- IntelliJ IDEA
  - Created in 2001
  - Russian company, headquarters is in St. Petersburg but large development center in Prague
  - Community vs Ultimate edition (Community is free, Ultimate is paid)
  - Ultimate edition price (2019): 12 290 Kč = ~482 € per year
  - In 2010 Sun Microsystems was acquired by Oracle => Netbeans is Oracle IDE now
  - Advantages:
    - (+) Feels context
    - (+) String checking for, e.g., JPQL queries, the best refactoring, fast
  - Disadvantages
    - (-) It is not free in full version
    - (-) It can be challenging for computers with RAM less than 2GB (old notebooks etc.)



# Integrated Development Environments (IDE)

- Final thoughts:
  - <https://pypl.github.io/IDE.html>
- In this course we will use:
  - IntelliJ IDEA Ultimate

**Worldwide**, Visual Studio is the most popular IDE, Android Studio grew the most in the last 5 years (18.4%) and Eclipse lost the most (-18.2%)





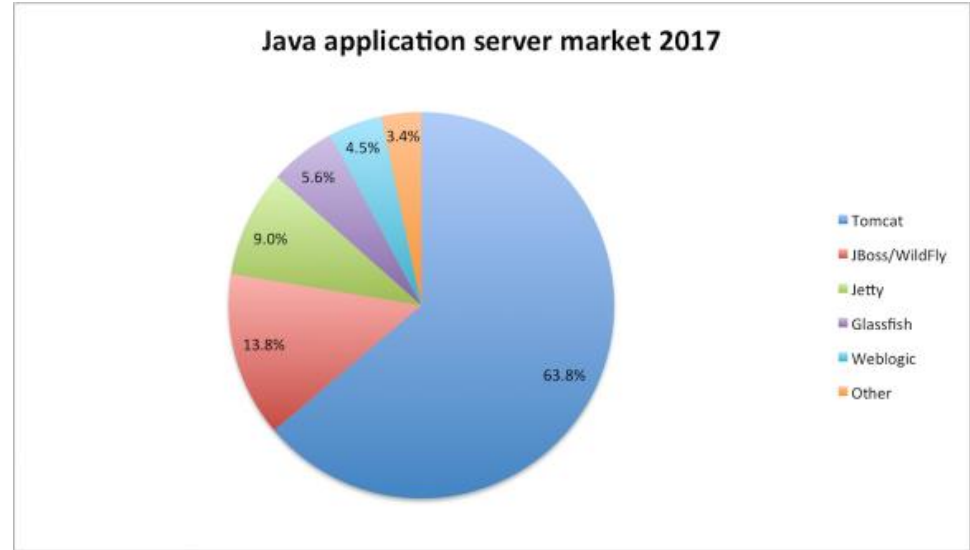
# Review of Application Servers

- What is application server?
  - Software framework that consists web server connectors, computer programming languages, runtime libraries, database connectors, and the administration code needed to deploy, configure, manage, and connect these components on a web host
- Web Servers:
  - Tomcat
  - Jetty
- Application Servers:
  - Glassfish (reference Java EE implementation)
  - Oracle Weblogic
  - IBM WebSphere
  - Wildfly
  - Apache Geronimo

Web Servers	Application Servers
Servlets, JSP, Web Socket	Servlets, JSP, Web Socket
Expression Language, JTA	Expression Language, JTA
	Batch, CDI, EJB, JPA, JMS
	JAX-RS, JAX-WS, JavaMail
	JSON-P, Bean Validation

# Review of Application Servers - Popularity

- Currently, Tomcat, WildFly and Jetty are the most popular (in 2017)
- The reason is probably the following:
  - They are easy to configure and manage
  - Tomcat and Jetty are well suited with Spring framework
- Web servers are commonly used these days as embedded servers inside .jar file





# Review of Application Servers - Selection

- How to choose web/application server?
  - With Spring basically the most common is to use Tomcat (default option as embedded server inside .jar)
  - With Jakarta/Java EE applications the most common is to use WildFly because it is free and quite easy to configure
- What aspects are important:
  - Number of concurrent users the system should support
  - IDE tooling and GUI support
  - Security and authentication
  - Technical support and documentation
  - How they implement Jakarta/Java EE specification
    - E.g., Glassfish implements JPA with EclipseLink and WildFly implements JPA with Hibernate framework
  - What extensions from Jakarta/Java EE specifications they are supporting, e.g. Weblogic supports MakeConnection for JAX-WS services



# Review of Application Servers

- Embedded vs external java web servers
    - Application with embedded server looks like a regular java program. You just launch jar file and that's it.
    - Regular web application is usually a war archive which needs to be deployed to some server
  - With embedded server your application is packaged with the server of choice and responsible for server start-up and management
- 
- |                                                                                                                                                                                                                                                                                                                                                                                               |                                                                                                                                                                                                                                                                                                                           |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>● Embedded pros:<ul style="list-style-type: none"><li>○ Single object to be deployed</li><li>○ You can easily test against server versions just like any other dependency</li></ul></li><li>● Embedded cons:<ul style="list-style-type: none"><li>○ Dependency bloat, as you have to include all the dependencies of the web server</li></ul></li></ul> | <ul style="list-style-type: none"><li>● External servers pros:<ul style="list-style-type: none"><li>○ Potentially more flexible application architecture</li></ul></li><li>● External servers cons:<ul style="list-style-type: none"><li>○ Deployment complexity</li><li>○ More difficult development</li></ul></li></ul> |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

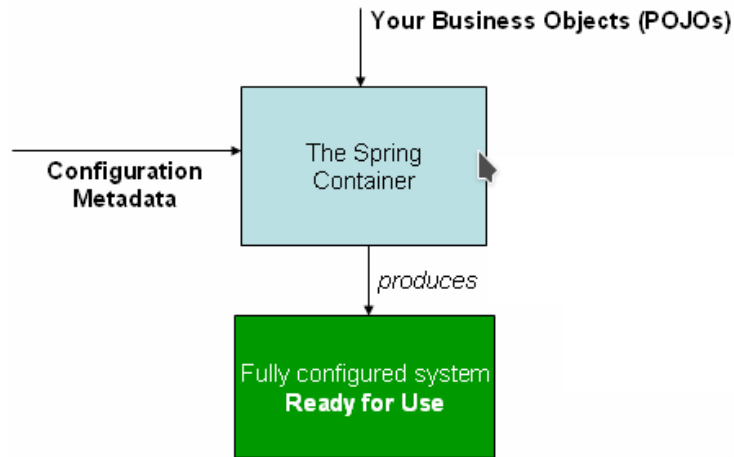


# Spring Framework vs Jakarta/Java EE

- Jakarta/Java EE
  - Is a specification for creating enterprise level Java applications
  - Basically includes a set of interfaces which are implemented by application server incorporates JDBC for databases, JNDI for registries, JTA for exchanges, JMS for informing, etc.
  - In 2018 moved from Oracle to Eclipse Foundation (Java EE renamed to Jakarta EE)
- Spring framework
  - Was created in October 2002 by Rod Johnson who wrote his book 'Expert One-on-One J2EE Design and Development'
  - It is complementary to Jakarta EE, it integrates with carefully selected individual specifications from the EE umbrella: Servlet API, WebSocket API, Concurrency Utilities, JSON Binding API, Bean Validation, JPA, JMS, JTA
  - Contains a large set of modules: <https://spring.io/projects>
- In personal opinion, Spring (Boot) is nowadays the winner for cloud-based microservices
  - It is also easier to develop particular things as Security
  - Also, the middleware (web/application server) management is usually easier

# Inversion of Control

- Spring container uses the design pattern of Inversion of Control (IoC) also known as dependency injection (DI)
- It is a process whereby objects define their dependencies, that is, the other objects they work with, only through constructor arguments, arguments to a factory method, or properties that are set on the object instance after it is constructed or returned from a factory method
- The container then injects those dependencies when it creates the bean. This process is fundamentally the inverse, hence the name Inversion of Control (IoC), of the bean itself controlling the instantiation or location of its dependencies by using direct construction of classes, or a mechanism such as the Service Locator pattern.





# Metadata Configurations

- There exist three ways of metadata configurations:
  1. XML
  2. Annotations
  3. Code driven (Java Config)





# Metadata Configurations - XML

- Configured by XML
- Different modules (different namespaces)
- Benefits
  - Pure declarative approach
  - Allows to change system configuration without changing code (very useful for customizations)
- Weaknesses
  - Not transparent
  - Lots of configuration is needed
  - Hard to maintain
  - Problem with refactoring



# Metadata Configurations - XML (Examples)

- The example of defining Spring bean (managed class by Spring container) using XML config

```
<bean id="personService" class ="com.sedaq.service.PersonService"/>
```

- The id attribute is a string that you use to identify the individual bean definition
  - Must be unique
- The class attribute defines the type of the bean and uses the fully qualified class name
- Bean naming convention is standard Java convention for instance field names

```
<bean id="petStore"  
    class="org.springframework.samples.jpetsy.store.services.PetStoreServiceImpl">  
    <property name="accountDao" ref="accountDao"/>  
    <property name="itemDao" ref="itemDao"/>  
    <!-- additional collaborators and configuration for this bean go here -->  
</bean>
```



# Metadata Configurations - XML (Examples)

- Composing XML-based configuration metadata

```
<beans>
  <import resource="services.xml"/>
  <import resource="resources/messageSource.xml"/>
  <import resource="/resources/themeSource.xml"/>
  <bean id="bean1" class="..." />
  <bean id="bean2" class="..." />
</beans>
```

- In the preceding example, external bean definitions are loaded from three files:
  - services.xml, messageSource.xml, themeSource.xml



# Metadata Configurations - Annotations

- Configured by annotations
- Benefits:
  - Less configuration needed
  - More clear and transparent
  - No problem with refactoring
- Weaknesses:
  - Less flexibility
- Can be combined with xml configuration: (<context:annotation-config/>, <context:component-scan

```
@Configuration
public class RepositoryConfig {
    private @Autowired DataSource dataSource;
    public @Bean AccountRepository accountRepository() {
        return new JdbcAccountRepository(dataSource);
    }
}
```



# Metadata Configurations - Annotations

- Proprietary
  - @Component, @Service, @Repository
  - @Autowired, @Required
- JSR-330:
  - @Named
  - @Inject, @Qualifier
- Other:
  - @Resource, @PersistenceContext, @PersistenceUnit

```
@Component  
public class MyClass{  
  
}
```



# Metadata Configurations - Code Driven

- Benefits
  - Almost any code could be evaluated during initialization
  - No problem with refactoring
  - Almost everything is configured in Java classes
- Weaknesses:
  - Pure imperative approach
  - Configuration is hardcoded into the class
- @Configuration, @Bean, @ComponentScan, @PropertySource, @Import
- Each @Bean method is evaluated just once!

```
@Configuration
public class RepositoryConfig {
    private @Autowired DataSource dataSource;
    public @Bean AccountRepository accountRepository() {
        return new JdbcAccountRepository(dataSource);
    }
}
```



# Move to Code Driven With Legacy XML Config

- It is possible to use `@ImportResource` to import XML configuration we do not know how to do in Java Config and other things just configure in Java classes

```
@Configuration
@ImportResource("classpath:applicationContext.xml")
@ComponentScan("com.sedaq")
public class Application{

    public static void main(String[] args){
        AnnotationConfigApplicationContext applicationContext =
            new AnnotationConfigApplicationContext(Application.class);
    }
}
```



# Metadata Configurations - How to Select?

- XML configuration has no restrictions but is more difficult to create and maintain (nowadays, not usual)
- In pure Spring annotations and XML configuration was the most usual one
- In Spring Boot annotations + code driven are the most usual one
- The trend is to move to Spring Boot, so the annotations + code driven are nowadays the preferred one
  - The reason is that Java developers are more familiar with Java than with difficult XML configurations

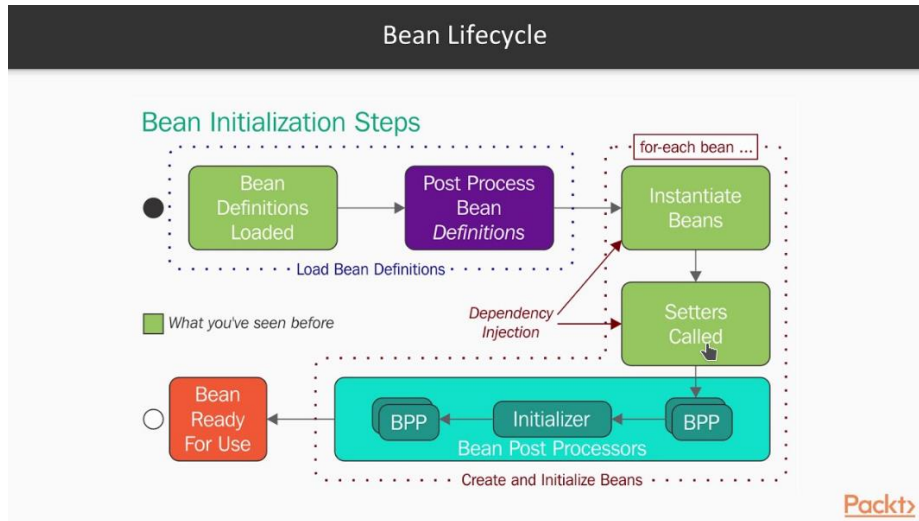




# Constructor, Setter or Property Based Injection

- Constructor injection is the best practice:
  - In object-oriented programming and object must be in a valid state after construction and every method invocation changes the state to another valid state
- Overriding: Setter injection overrides the constructor injection. If we use both constructor and setter injection, IoC container will use the setter injection
- Prefer constructor injection to make code cleaner

# Spring Bean Lifecycle



- If the Spring context is created all the declared beans are initialized (it is called eager loading)
- If the Spring context is shut down all the beans are destroyed from memory

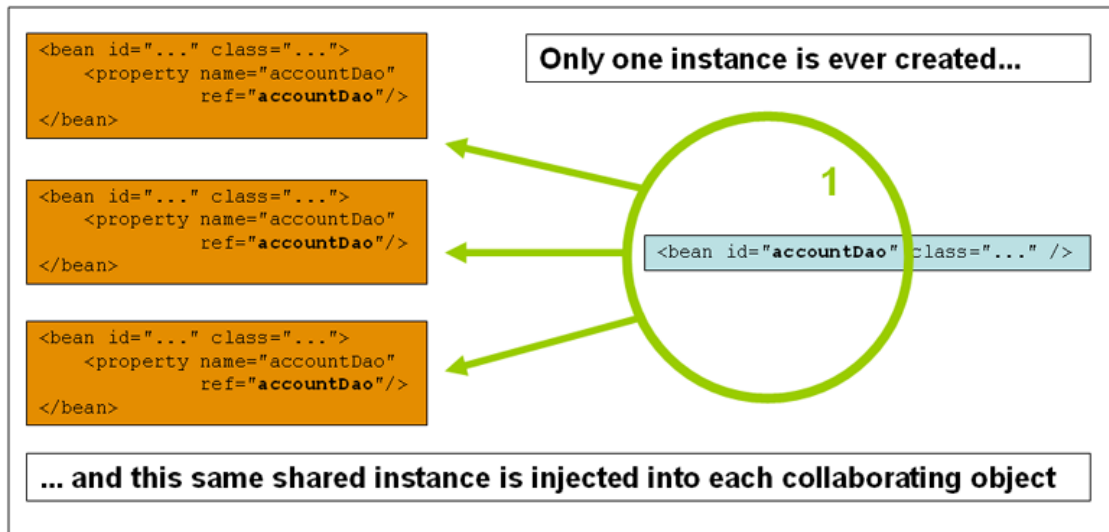
# Spring Bean Scopes



Scope	Description
singleton	(Default) Scopes a single bean definition to a single object instance per Spring IoC container.
prototype	Scopes a single bean definition to any number of object instances.
request	Scopes a single bean definition to the lifecycle of a single HTTP request; that is, each HTTP request has its own instance of a bean created off the back of a single bean definition. Only valid in the context of a web-aware Spring ApplicationContext.
session	Scopes a single bean definition to the lifecycle of an HTTP Session. Only valid in the context of a web-aware Spring ApplicationContext.
application	Scopes a single bean definition to the lifecycle of a ServletContext. Only valid in the context of a web-aware Spring ApplicationContext.

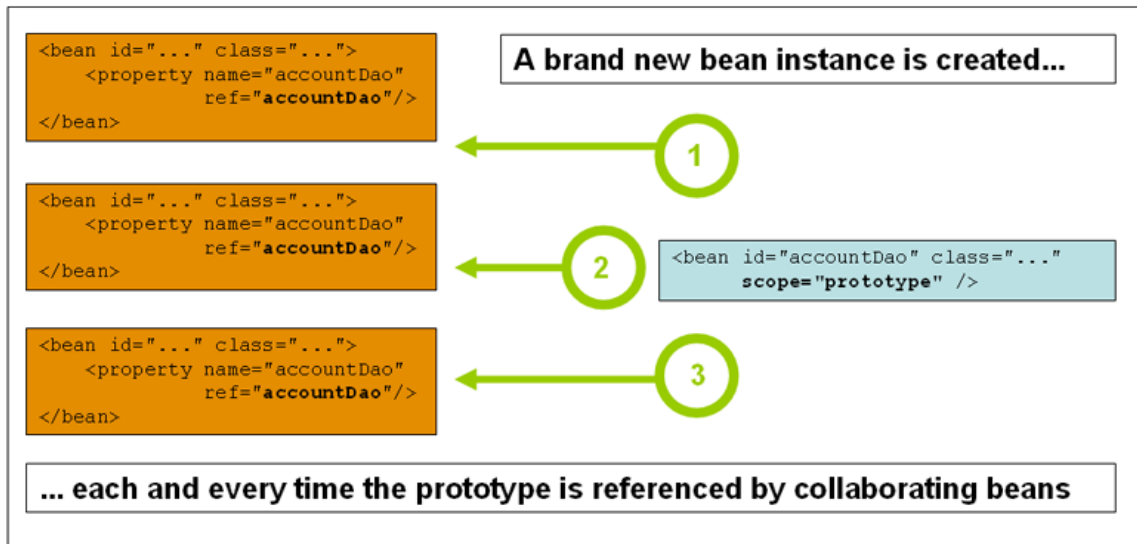
# Spring Bean Scopes - Singleton

- Only one shared instance of a singleton bean is managed, and all requests for beans with an id or ids matching that bean definition result in that one specific bean instance being returned by the Spring container
- Spring singleton is best described as per container and per bean



# Spring Bean Scopes - Prototype

- The non-singleton, prototype scope of bean deployment results in the *creation of a new bean instance* every time a request for that specific bean is made
- As a rule, use the *prototype* scope for all stateful beans and the *singleton* scope for stateless beans



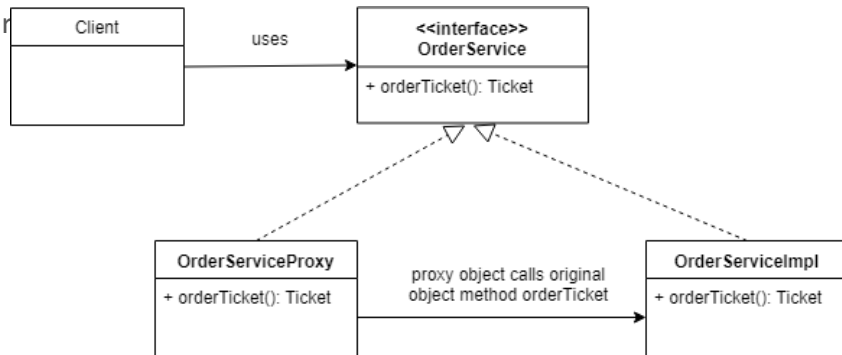


# Spring Bean Scopes - Real World Scenarios

- **Singleton:** Connection to a database
- **Prototype:** Declare configured form elements (a textbox configured to validate names, e-mail addresses for example) and get “living” instances of them for every form being created
- **Request:** information that should only be valid on one page like the result of a search or the confirmation of an order. The bean will be valid until the page is reloaded
- **Session:** To hold authentication information getting invalidated when the session is closed (by timeout or logout). You can store other user information that you do not want to reload with every request here as well

# Proxy Objects

- Proxy design pattern falls under the structural design pattern category and it is one of the most frequently used pattern in software development
- This pattern helps to control the usage and access behaviour of connected resources
- The proxy is the object that is being called by the client to access the real object behind the scene
- Gang of Four (GoF) defines it as:
  - “Provide a sur





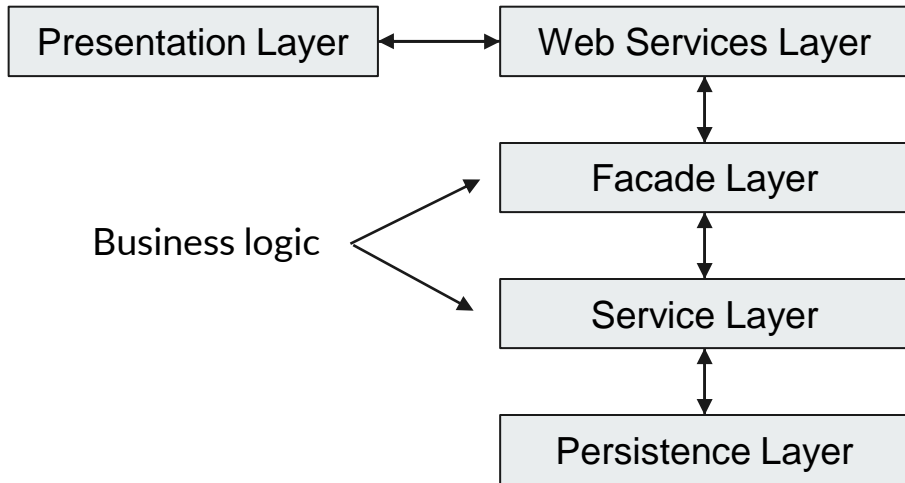
# Reviewing of Architecture Styles (Monolithic, ...)

- Monolithic means that all the code is composed in one piece
- Monolithic software is designed to be self-contained; components of the program are interconnected and interdependent rather than loosely coupled
- Benefits:
  - Easier to design
  - For smaller applications it is preferred way
  - It usually has better throughput than modular approaches, such as the microservice architecture
  - Easier transaction management (distributed transactions are quite a hell)
- Cons
  - If any program component must be updated, the whole application has to be rewritten
  - Difficult for continuous delivery and deployment
  - Compiling and building large projects takes a lot of time
  - Developers are not able to create their own applications, they are just learning the project as it is



# Reviewing of Architecture Styles (n-tier, ...)

- N-tier applications are usually divided into a few layers, these includes:



- Benefits:
  - Easy to manage: You can manage each tier separately, adding or modifying each tier without affecting the other tiers
  - More efficient development: You can split developer teams, easier to find particular logic in the code
- Disadvantage:
  - The performance could be slower



# Reviewing of Architecture Styles (Microservices, ...)

- Used by Netflix, Amazon, eBay
- Microservice architecture is basically a set of loosely coupled, collaborating services
- Benefits:
  - Each service is highly maintainable and testable - enable rapid and frequent development and deployment
  - Loosely coupled with other services - enables a team to work independently the majority of time on their service(s) without being impacted by changes to other services and without affecting other services
  - Independently deployable - enables a team to deploy their service without having to coordinate with other teams
  - Capable of being developed by a small team - essential for high productivity by avoiding the high communication head of large teams
- Drawbacks:
  - Additional complexity with creating a distributed systems (dealing with partial failures, distributed transactions)
  - Testing the interactions between services is more difficult
  - Developers need to learn a huge set of difficult principles
  - Network bandwidth is higher (microservices could be deployed in other servers; multiple



## **Section: Spring and Spring Boot**



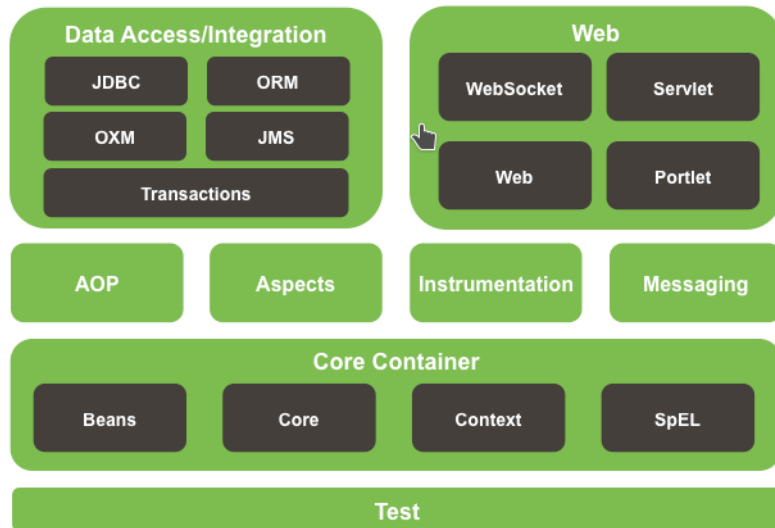
# Spring and Spring Boot

- Spring modules
- Spring
- Spring Boot autoconfiguration
- Spring Boot parent project
- Initializing a Spring application

# Spring Modules



Spring Framework Runtime





# Spring Modules

- Spring modules could be separated into several parts:
  - **Core Container** - contains the most important parts as IoC container
  - **Data Access/Integration** - contains easier working with database using JdbcTemplate (wrapper for plain JDBC), or Spring Data which is wrapper for JPA
  - **Web** - which is used for easier web applications and web services (REST services) development
  - **Test** - the packages for implementing unit and integration tests
  - **AOP, Aspects, Instrumentation, Messaging**



# Spring Applications Deployment

- Nowadays, Spring applications are widely used in the industry for cloud-based applications
- The usage of Spring framework is almost everywhere:
  - Banking systems
  - Streaming platforms
  - Applications of government agendas
  - Telecommunication systems (especially module Spring Integration)



# Spring is Not Only Framework

- In Spring a lot of subprojects were created:
  - <https://spring.io/projects>
  - **Spring Data** - easier working with several databases
  - **Spring Security** - the complete solution for security scenarios (OpenID Connect, OAuth2, authentication, authorization, etc.)
  - **Spring Batch** - data management
  - **Spring LDAP** - support for LDAP storages including transactions; LdapTemplate (similar to JdbcTemplate)
  - **Spring Web Services** - support for SOAP-based web services
  - **Spring Cloud** - support for microservice architectures (Eureka, Hystrix, Zuul, etc.)
  - **Spring Integration** - useful for telecommunication companies to implement protocols like MQTT, web sockets and so on
  - **Spring Web Flux** - The sweet spot for Spring Web Flow are stateful web applications with controlled navigation such as checking in for a flight, applying for a loan, shopping cart checkout, or even adding a confirmation step to a form





# Support for Spring in IDEs

- SpringSource Tool Suite (STS) - It is the extension of Eclipse IDE supported by Pivotal company (the owner of Spring framework)
- Spring IDE - the lightweight version of STS IDE
- IntelliJ IDEA
- Netbeans IDE

# Creating a Simple Spring Application

- The minimum required dependencies:

```
<dependencies>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-core</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
  </dependency>
</dependencies>
```

The core library, necessary for creating Spring IoC etc.

Basic library of Spring, suitable for “hello world” applications



# Creating a Simple Spring Application

- Starting up Spring container
  - Using XML plus annotations config (“applicationContext.xml”)
  - Placed in the root of classpath, e.g., in resources folder

```
<?xml version="1.0" encoding="UTF-8" ?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">

    <context:annotation-config></context:annotation-config>
    <context:component-scan base-package="com.sedaq"/>

    <!-- <bean id="userRepository" class="com.sedaq.repository.UserRepository"/> -->
</beans>
```

# Creating a Simple Spring Application

- Creating and starting Spring container
- Based on ClassPathXmlApplicationContext:

Loading XML config on classpath

```
import com.sedaq.model.Person;
import com.sedaq.repository.UserRepository;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class AppXmlAppContext {

    public static void main(String[] args) {
        ClassPathXmlApplicationContext applicationContext =
            new ClassPathXmlApplicationContext( configLocation: "applicationContext.xml");
        UserRepository userRepository = applicationContext.getBean(UserRepository.class);
        // call methods on userRepository
        Person person = userRepository.getPersonById(1L);
        System.out.println(person);
        applicationContext.close();
    }
}
```



# Creating a Simple Spring Application

- Creating and starting Spring container
- Based on AnnotationConfigApplicationContext:

```
@Configuration
@ImportResource("classpath:applicationContext.xml")
@ComponentScan("com.sedag")
public class App {

    public static void main(String[] args) {
        AnnotationConfigApplicationContext applicationContext = new AnnotationConfigApplicationContext(App.class);
        UserRepository userRepository = applicationContext.getBean(UserRepository.class);
        // call methods on userRepository
        Person person = userRepository.getPersonById(1L);
        System.out.println(person);
        applicationContext.close();
    }
}
```

← Loading "legacy" XML config



# Creating a Simple Spring Boot Application

- Maven dependencies:

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.1.6.RELEASE</version>
  <relativePath/> <!-- lookup parent from repository -->
</parent>
```

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>

<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
```



# Creating a Simple Spring Boot Application

- Creating and starting Spring container IoC
- As you can see, with Spring Boot it is quite easy
- ... and it's only the beginning of this framework in framework..

```
@SpringBootApplication
public class App {

    public static void main(String[] args) {
        ApplicationContext applicationContext = SpringApplication.run(App.class, args);
        UserRepository userRepository = applicationContext.getBean(UserRepository.class);
        userRepository.getPersonById(1L);
    }
}
```

# Spring Boot Bundles

- We have only two dependencies in Maven and so much external libraries were downloaded...
- What happened?

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>

<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
```

External Libraries

- > < 11 > C:\Program Files\Java\jdk-11
- > Maven: ch.qos.logback:logback-classic:1.2.3
- > Maven: ch.qos.logback:logback-core:1.2.3
- > Maven: com.jayway.jsonpath:json-path:2.4.0
- > Maven: com.vaadin.external.google:android-json:0.0.20131108.vaadin1
- > Maven: javax.annotation:javax.annotation-api:1.3.2
- > Maven: junit:junit:4.12
- > Maven: net.bytebuddy:byte-buddy:1.9.13
- > Maven: net.bytebuddy:byte-buddy-agent:1.9.13
- > Maven: net.minidev:accessors-smart:1.2
- > Maven: net.minidev:json-smart:2.3
- > Maven: org.apache.logging.log4j:log4j-api:2.11.2
- > Maven: org.apache.logging.log4j:log4j-to-slf4j:2.11.2
- > Maven: org.assertj:assertj-core:3.11.1
- > Maven: org.hamcrest:hamcrest-core:1.3
- > Maven: org.hamcrest:hamcrest-library:1.3
- > Maven: org.mockito:mockito-core:2.23.4
- > Maven: org.objenesis:objenesis:2.6
- > Maven: org.ow2.asm:asm:5.0.4
- > Maven: org.skyscreamer:jsonassert:1.5.0
- > Maven: org.slf4j:jul-to-slf4j:1.7.26
- > Maven: org.slf4j:slf4j-api:1.7.26
- > Maven: org.springframework.boot:spring-boot:2.1.6.RELEASE
- > Maven: org.springframework.boot:spring-boot-autoconfigure:2.1.6.RELEASE
- > Maven: org.springframework.boot:spring-boot-starter:2.1.6.RELEASE
- > Maven: org.springframework.boot:spring-boot-starter-logging:2.1.6.RELEASE
- > Maven: org.springframework.boot:spring-boot-starter-test:2.1.6.RELEASE
- > Maven: org.springframework.boot:spring-boot-test:2.1.6.RELEASE
- > Maven: org.springframework.boot:spring-boot-test-autoconfigure:2.1.6.RELEASE
- > Maven: org.springframework:spring-aop:5.1.8.RELEASE
- > Maven: org.springframework:spring-beans:5.1.8.RELEASE
- > Maven: org.springframework:spring-context:5.1.8.RELEASE
- > Maven: org.springframework:spring-core:5.1.8.RELEASE
- > Maven: org.springframework:spring-expression:5.1.8.RELEASE
- > Maven: org.springframework:spring-jcl:5.1.8.RELEASE
- > Maven: org.springframework:spring-test:5.1.8.RELEASE
- > Maven: org.xmlunit:xmlunit-core:2.6.2
- > Maven: org.yaml:snakeyaml:1.23





# Why Use Spring Boot?

- To ease the Java-based applications development, unit and integration tests process
- To reduce development time by providing some defaults
- To increase productivity
- Uses best practices in the project creation
- No XML, configured by annotations and Java config
- Autoconfiguration is amazing
- Is Spring Boot too much magic?
  - Craig Walls statement: “When I drive my car, it’s unnecessary for me to fully understand how the engine works or individual components that the engine is made up from. Certainly, there are some people who enjoy that kind of detail, but for me it’s sufficient to turn the key, pump the gas, and steer my vehicle from one place to another. I take advantage of the abstraction (ignition, steering wheel, pedals) trusting that they properly coordinate with the underlying mechanics of the automobile.”



# Spring Boot Executable .jar File (The Default)

- By default Spring Boot uses as final executable file the .jar file instead of .war
- To create such a class it is necessary to have the following class

```
@SpringBootApplication
public class App {

    public static void main(String[] args) {
        SpringApplication.run(App.class, args);
    }

}
```

- By default Spring Boot uses as final executable file the .jar file instead of .war
- To create such a class it is necessary to have the following class

- To enable what auto-configurations are enabled start the application with --debug
- By default Spring Boot uses embedded Tomcat



## **Section: Database Access**



# Database Access

- Database Access
  - a. Java Database Connectivity (JDBC)
  - b. Spring JDBC Template
  - c. JPA basics:
    - i. Entities, Entity relationships,
    - ii. Common annotations (@Entity, @Table, @Column, @Id, etc.)
    - iii. Fetch types (Lazy vs Eager)
    - iv. EntityManager
  - d. Spring Data
    - i. Paging and sorting
  - e. Transactions

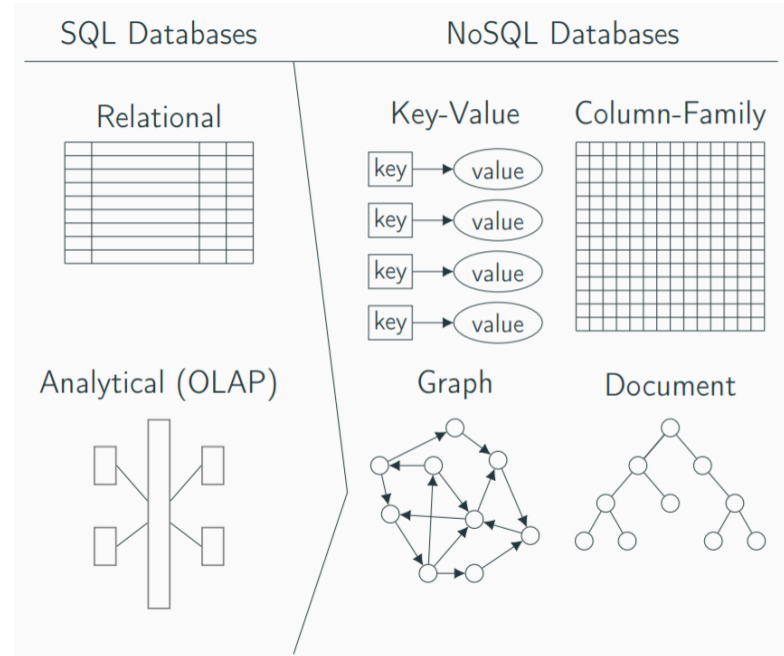


# Database Access - Types of Databases

- Relational databases
  - Primarily used for storing and manipulation with structured data
  - Based on the relational data model created by Frank F. Codd in 1970 in IBM labs
  - Usually follows high level of consistency through Atomicity Consistency Isolation Durability (ACID)
  - The database design commonly follows the principle of normal forms (1NF, 2NF, 3NF, ...)
  - The examples: Oracle DB, IBM DB2, PostgreSQL, MySQL, Microsoft SQL Server, ...
- NoSQL databases
  - Usually used for storing semi-structured or unstructured data
  - Usually managed by a Consistency Availability Partition tolerance (CAP) or a Basically Available Soft-State Eventual consistency (BASE)
  - Data consistency is not crucial... good for marketing and behaviour data
  - The examples: MongoDB, Cassandra, Redis, ...
- <https://db-engines.com/en/ranking>

# Database Access - NoSQL Databases

- NoSQL databases are divided into four main categories:
  - Key-Value -> Redis, ...
  - Column-Family -> Cassandra, ...
  - Graph -> Neo4J, ...
  - Document -> MongoDB, ...
- For these types of databases special query methods are essential
- How to work with them in Spring?
- -> Spring Data contains support for several NoSQL databases, e.g., MongoDB, Redis, Cassandra, Neo4J

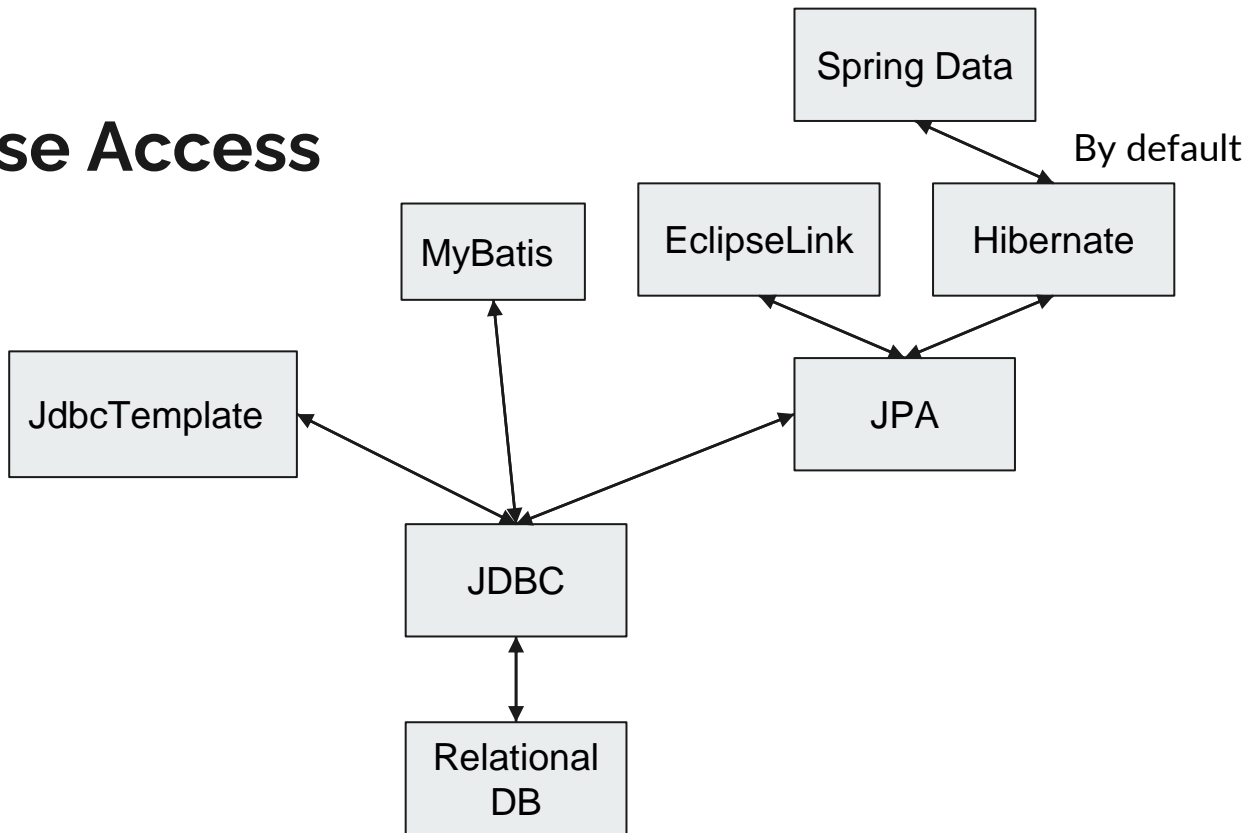




# Database Access - Relational Databases

- Relational databases are designed using normal forms with the goal to reduce data redundancy and improve data integrity.
  - 1NF: To satisfy 1NF, we need to ensure that the values in each column of a table are atomic
  - 2NF:
    - Must be in 1NF
    - It does not have any non-prime attribute that is functionally dependent on any proper subset of any candidate key of the relation. A non-prime attribute of a relation is an attribute that is not a part of any candidate key of the relation.
  - 3NF:
    - Must be in 2NF
    - No non-prime (non-key) attribute is transitively dependent on any key i.e. no non-prime attribute depends on other non-prime attributes. All the non-prime attributes must depend only on the keys.
- Sometimes for analytical purposes it is suitable to do database denormalization

# Database Access







# Database Access

- JDBC
  - Accesses data as rows and columns
- JPA
  - Accesses data through Java objects using a concept called object-relational mapping (ORM). The idea is that you don't have to write as much code, and you get your data in Java objects.
- Each principle has it's cons and pros

# Identifying the Structure of Relational Database

**Table** also called **Relation**

Primary Key

Domain  
Ex: NOT NULL

© guru99.com

CustomerID	CustomerName	Status
1	Google	Active
2	Amazon	Active
3	Apple	Inactive

**Tuple OR Row**  
Total # of rows is **Cardinality**

**Column OR Attributes**  
Total # of column is **Degree**



# Relational Database - Concepts

- **Attribute:** Each column in a Table. Attributes are the properties which define a relation. e.g., Student\_Rollno, NAME,etc.
- **Tables** – In the Relational model the, relations are saved in the table format. It is stored along with its entities. A table has two properties rows and columns. Rows represent records and columns represent attributes.
- **Tuple** – It is nothing but a single row of a table, which contains a single record.
- **Relation Schema:** A relation schema represents the name of the relation with its attributes.
- **Degree:** The total number of attributes which in the relation is called the degree of the relation.
- **Cardinality:** Total number of rows present in the Table.
- **Column:** The column represents the set of values for a specific attribute.
- **Relation instance** – Relation instance is a finite set of tuples in the RDBMS system. Relation instances never have duplicate tuples.
- **Relation key** - Every row has one, two or multiple attributes, which is called relation key.
- **Attribute domain** – Every attribute has some pre-defined value and scope which is known as attribute domain



# Relational Model - Advantages

- **Simplicity:** A relational data model is simpler than the hierarchical and network model.
- **Structural Independence:** The relational database is only concerned with data and not with a structure. This can improve the performance of the model.
- **Easy to use:** The relational model is easy as tables consisting of rows and columns is quite natural and simple to understand
- **Query capability:** It makes possible for a high-level query language like SQL to avoid complex database navigation.
- **Data independence:** The structure of a database can be changed without having to change any application.
- **Scalable:** Regarding a number of records, or rows, and the number of fields, a database should be enlarged to enhance its usability.



# Relational Model - Disadvantages

- Few relational databases have limits on field lengths which can't be exceeded.
- Relational databases can sometimes become complex as the amount of data grows, and the relations between pieces of data become more complicated.
- Complex relational database systems may lead to isolated databases where the information cannot be shared from one system to another.
- Difficult working with BLOB data



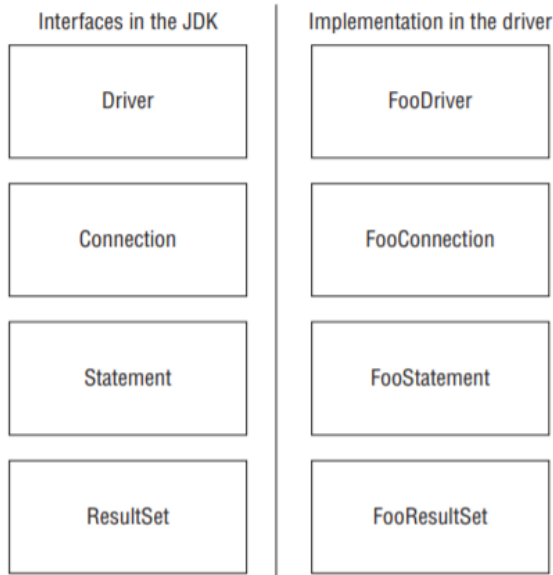
# Writing Basic SQL Statements

- **INSERT:** Add a new row to the table
  - **SELECT:** Retrieve data from the table
  - **UPDATE:** Change zero or more rows in the table
  - **DELETE:** Remove zero or more rows from the table
- 
- The example:
    - `SELECT id, email FROM person`



# Java Database Connectivity (JDBC)

- Introducing the interfaces of JDBC
- What these drivers do?
- **Driver:** Knows how to get a connection to the database
- **Connection:** Knows how to communicate with the database
- **Statement:** Knows how to run the SQL
- **ResultSet:** Knows what was returned by a SELECT query



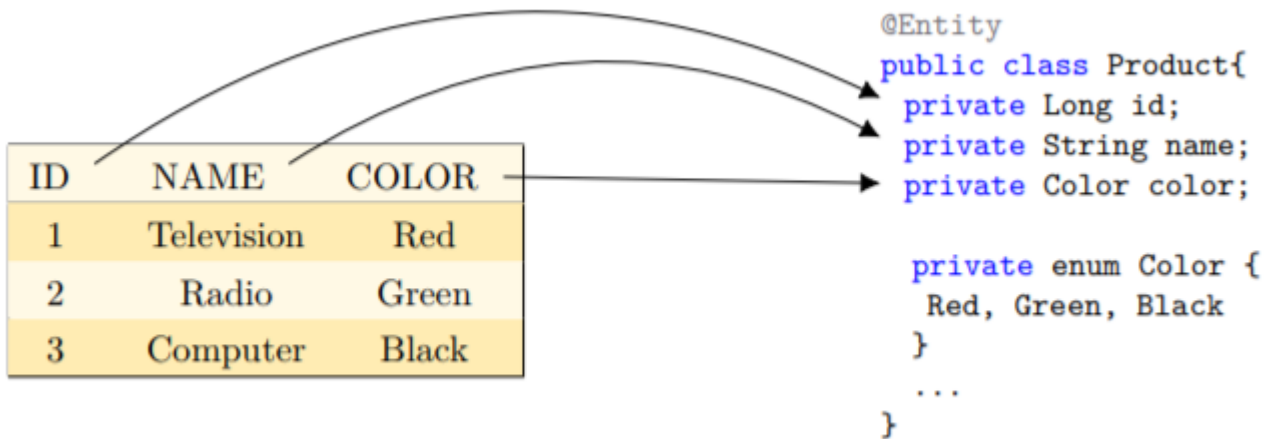


# Java Persistence API (JPA)

- Replaces EJB 2.1 entity beans with non EJB-entity classes
- It is specification for Object Relational Mapping (ORM)
- It does not require Java/Jakarta EE servers (Websphere, Weblogic, Tomcat, ...)
- Can be used with:
  - Container-managed persistence
  - Application-managed persistence
- JPA provides and object representation of a data
  - Entity classes are mapped to a relational database
- Several frameworks are implementing this specification
  - EclipseLink (the reference implementation), Hibernate (by default used by Spring), TopLink etc.



# JPA





# JPA Entity

- Plain Old Java Object (POJO) class
- This class must contain non-argument constructor and @Entity annotation
- Is serializable, can be used as detached object
- Entities can be queried
- It can be uniquely identified by a primary key
- It manages persistent data in concern with a JPA entity manager



# Entity Class Requirements

- The class must be annotated with the `javax.persistence.Entity` annotation
- The class must have a public or protected, no-argument constructor. The class may have other constructors
- The class must not be declared final. No methods or persistent instance variables must be declared final
- If an entity instance is passed by value as a detached object, such as through a session bean's remote business interface, the class must implement the `Serializable` interface.
- Entities may extend both entity and non-entity classes, and non-entity classes may extend entity classes
- Persistent instance variables must be declared private, protected, or package-private and can be accessed directly only by the entity class's methods. Clients must access the entity's state through accessor or business methods



# JPA - Creating an Person Entity

`@Entity`

Defines Entity

```
public class Person implements Serializable {
```

`@Id`

Primary Key

```
@GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
@Column(name = "id", updatable = false, nullable = false)
```

```
private Long id;
```

`@Column(nullable = true)`

Columns Definitions

```
private LocalDate birthday;
```

```
public Person() {
```

```
    // hibernate requires non-args constructor
```

```
}
```

```
}
```



# Entity Class - Primary Keys

- Every JPA entity must have a primary key
- It is used to distinguish one entity instance from another
  - It is used by EntityManager to identify the object
  - The @Id annotation is used to identify the entity primary key
  - Commonly, primary key is set as Long or Integer, but it can be a class that corresponds to several database columns (composite primary keys)

@Id private Long id;

- Primary keys could be auto-generated (the most common variant)



# Entity Class - Primary Keys - Generation Strategies

- JPA provides four generation strategies
  - AUTO
  - IDENTITY
  - SEQUENCE
  - TABLE
- The strategy is specified using @GeneratedValue annotation

```
@Id
@GeneratedValue(strategy = GenerationType.AUTO)
private Long id;
```



# Generation Strategy - AUTO

- The `GenerationType.AUTO` is the default generation type and lets the persistence provider choose the generation strategy.
- If you use Hibernate as your persistence provider, it selects a generation strategy based on the database specific dialect. For most popular databases, it selects `GenerationType.SEQUENCE`

```
@Id
@GeneratedValue(strategy = GenerationType.AUTO)
private Long id;
```



## Generation Strategy - IDENTITY

- The `GenerationType.IDENTITY` is the easiest to use but not the best one from a performance point of view
- It relies on an auto-incremented database column and lets the database generate a new value with each insert operation
- From a database point of view, this is very efficient because the auto-increment columns are highly optimized, and it doesn't require any additional statements

```
@Id @GeneratedValue(strategy = GenerationType.AUTO)
private Long id;
```

- This approach has a significant drawback if you use Hibernate. Hibernate requires a primary key value for each managed entity and therefore has to perform the insert statement immediately
- This prevents it from using different optimization techniques like JDBC batching.





# Generation Strategy - SEQUENCE

- It requires additional select statements to get the next value from a database sequence. But this has no performance impact for most applications
- And if your application has to persist a huge number of new entities, you can use some [Hibernate specific optimizations](#) to reduce the number of statements.

```
@Id @GeneratedValue(strategy = GenerationType.Sequence)
@SequenceGenerator(name="person_generator", sequenceName = "person_seq", allocationSize=50)
private Long id;
```



## Generation Strategy - TABLE

- The *GenerationType.TABLE* gets only rarely used nowadays
- It simulates a sequence by storing and updating its current value in a database table which requires the use of pessimistic locks which put all transactions into a sequential order
- This slows down your application, and you should, therefore, prefer the *GenerationType.SEQUENCE*, if your database supports sequences, which most popular databases do



# Generation Strategy - SUMMARY

JPA offers 4 different ways to generate primary key values:

1. AUTO: Hibernate selects the generation strategy based on the used dialect,
2. IDENTITY: Hibernate relies on an auto-incremented database column to generate the primary key,
3. SEQUENCE: Hibernate requests the primary key value from a database sequence,
4. TABLE: Hibernate uses a database table to simulate a sequence.



# Overriding Default Mapping

- When you design JPA entities you probably reach to state that you need to rename the table or column names
- It is done by @Table annotation on a class and @Column annotation on an field

```
@Entity
@Table(name = "contact")
public class Contact implements Serializable {
    ...
    @Column(name = "contact_type")
    private String contactType;
}
```

- If you need to escape the names you could use the following: `@Table(name = "\"contact\"")`



# Persistent Data Types

Properties can be of the following data types:

- Java primitive types and Java wrappers, including:
  - Integer, Boolean, Float, and Double
- Serializable types, including:
  - String, BigDecimal, and BigInteger
- Arrays of bytes and characters, including:
  - byte[] and Byte[], char[] and Character[]
- Temporal types, including (but not limited to):
  - java.util.Date, java.util.Calendar, java.sql.Date, and java.sql.Timestamp
  - Since Hibernate 5 it is supported to use LocalDateTime, .. and the other classes from package java.time
  - In Hibernate 4 you must apply converters for that purpose
- Enums and collections
- Other entities



# Persistent Fields Versus Persistent Properties

- In JPA, the state is retrieved from fields or from properties
- The entity state is synchronized with the database
- The base rules for field access is:
  - It should not be read from client directly (it should be private)
  - Unless they are annotated with `@Transient`, all the fields are persistent
- The base rules for property access is:
  - Access should be public or protected
  - Persistent annotation could be placed only on getter
  - It must follow the JavaBeans naming convention
- What access to choose?
  - It depends if we need additional process, If we need additional processing then property-based access is preferred, in other way the field access is preferred
  - If we annotate both field and property, on one of them should be `@Transient` annotation

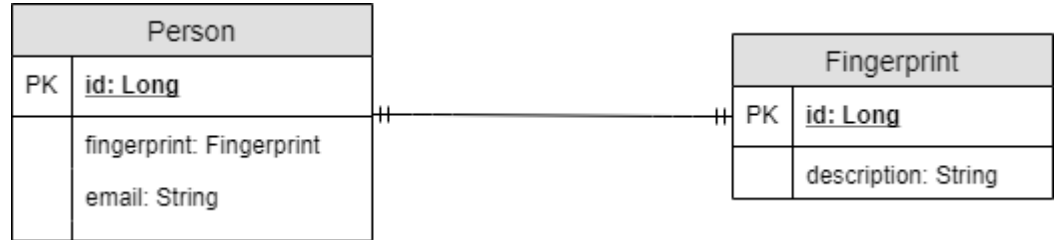


# JPA Database Schema Modelling

- Until now, we used only single entity classes
- In practice, databases have a lot of tables
- For that reason, JPA provides the possibility to design relations between entities
- The relations are basically:
  - Uni-directional (the relation is presented only at its owning side)
  - Bi-directional (the relation is presented in both sides)
- These relations could be one-to-one, one-to-many, many-to-one, many-to-many

# JPA Database Schema Modelling - One-to-One

- Each person has directly one fingerprint and each fingerprint is assigned to directly one person
  - “Hopefully”
- Person is the owning side
- Owning side means that it holds the Foreign Key (FK)





# One-to-One Relation Using uni-directional

```
@Entity
public class Person {
    @Id
    private Long id;
    private String email;
    @OneToOne
    @JoinColumn(name="fingerprint_id")
    private Fingerprint fingerprint;
}
```

```
@Entity
public class Fingerprint {
    @Id
    private Long id;
    private String description;
}
```



# One-to-One Relation Using bi-directional

```
@Entity
public class Person {
    @Id
    private Long id;
    private String email;
    @OneToOne
    @JoinColumn(name="fingerprint_id")
    private Fingerprint fingerprint;
}
```

Person	
PK	<u>id: Long</u>
	fingerprint: Fingerprint
	email: String

```
@Entity
public class Fingerprint {
    @Id
    private Long id;
    private String description;
    @OneToOne(mappedBy="fingerprint")
    private Person person;
}
```

Fingerprint	
PK	<u>id: Long</u>
	description: String

Name of attribute in reference entity

This is called the inverse side

# One-to-Many Relation Using uni-directional

- The one side is always holding the FK



# One-to-Many Relation Using uni-directional

```
@Entity
public class Person {
    @Id
    private Long id;
    private String email;
    private String name;
}
```

```
@Entity
public class Order {
    @Id
    private Long id;
    @ManyToOne
    @JoinColumn(name = "person_id")
    private Person person;
}
```



# One-to-Many Relation Using bi-directional

```
@Entity
public class Person {
    @Id
    private Long id;
    private String email;
    private String name;
    @OneToMany(mappedBy = "person")
    private Set<Order> orders;
}
```

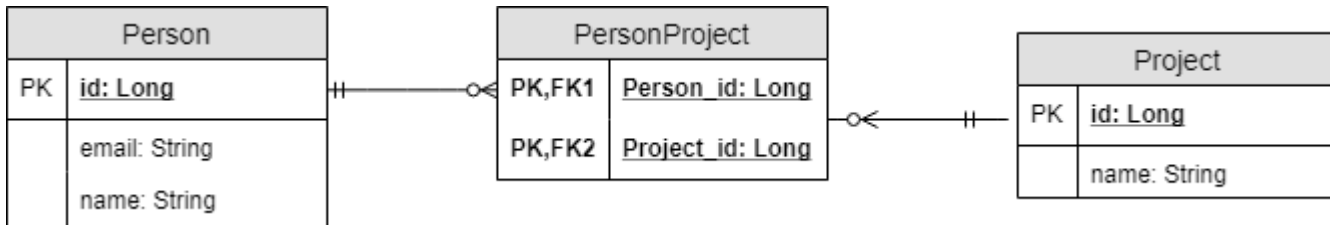
```
@Entity
public class Order {
    @Id
    private Long id;
    @ManyToOne
    @JoinColumn(name = "person_id")
    private Person person;
}
```



# Many-to-Many Relation Using bi-directional

```
@Entity
public class Person {
    @Id
    private Long id;
    @ManyToMany
    @JoinTable(name="PersonProject",
        joinColumns=@JoinColumn(name="Person_id"),
        inverseJoinColumns=@JoinColumn(name="Project_id"))
    private Set<Project> projects;
}
```

```
@Entity
public class Project {
    @Id
    private Long id;
    @ManyToMany(mappedBy = "projects")
    private Set<Person> persons;
}
```





## What Collections You Should Use?

- Prefer using Set for many-to-many associations
- Set is the most similar to the relational abstract model
- In some versions of Hibernate (e.g., in Hibernate 4) the List for collections generates additional queries



# FetchTypes

- The *FetchType* defines when JPA gets the related entities from the database, and it is one of the crucial elements for a fast persistence tier
- In general, you want to fetch the entities you use in your business tier as efficiently as possible. But that's not that easy.
- You either get all relationships with one query or you fetch only the root entity and initialize the relationships as soon as you need them.





# FetchType - LAZY

- Fetch type LAZY means that when you retrieve person entity no additional query for orders is processed
- If you press getOrders() then the additional query for retrieving orders for that person is executed

```
@Entity
public class Person {
    @Id
    private Long id;
    private String email;
    private String name;
    @OneToMany(mappedBy = "person", fetch = FetchType.LAZY)
    private Set<Order> orders;
}
```



# FetchType - EAGER

- Fetch type EAGER means that the related entity is loaded into memory at build-time
- This is quite dangerous in case you have a lot of data..
- It could be used only in cases when you always want to work with all the data from related entity

```
@Entity
public class Person {
    @Id
    private Long id;
    private String email;
    private String name;
    @OneToMany(mappedBy = "person", fetch = FetchType.EAGER)
    private Set<Order> orders;
}
```



# Default FetchType

- In JPA 2.0:
  - OneToMany: LAZY
  - ManyToOne: EAGER
  - ManyToMany: LAZY
  - OneToOne: EAGER



# FetchType LAZY N+1 Problem

- Consider the following scenario?
- How many SQL commands are executed?

```
@Entity
public class Person {
    @Id
    private Long id;
    private String email;
    private String name;
    @OneToMany(mappedBy = "person", fetch = FetchType.LAZY)
    private Set<Order> orders;

    public Set<Order> getOrders(){ return orders;}
}
```

```
TypedQuery<Person> persons = em.query("SELECT p FROM
Person p", Person.class);
List<Person> personsList = em.getResultList();
personsList.forEach(person ->{
    System.out.println(person.getOrders());
});
```

# Cascading

- Removing an entity in a relationship

```
public void removeFingerPrint(Long personId) {  
    Person person = em.find(Person.class, personId);  
    FingerPrint fingerPrint = person.getFingerPrint();  
    em.remove(fingerPrint);  
}
```

- This will lead to the database error foreign key violation
- To fix it we must explicitly remove fingerPrint from and Person before the commit



```
public void removeFingerPrint(Long personId) {  
    Person person = em.find(Person.class, personId);  
    FingerPrint fingerPrint = person.getFingerPrint();  
    person.setFingerPrint(null);  
    em.remove(fingerPrint);  
}
```



# Cascading

- Entities with other entities could be managed by cascading operations

```
@Entity
public class Person {
    @Id
    private Long id;
    private String email;
    @OneToOne(cascade = CascadeType.REMOVE)
    @JoinColumn(name="fingerprint_id")
    private Fingerprint fingerprint;
}
```

Cascade mode attributes include:

- **PERSIST**: Cascade persist operations from source to target.
- **MERGE**: Cascade merge operations.
- **REMOVE**: Cascade remove operations.
- **REFRESH**: Cascade refresh operations.
- **DETACH**: Cascade detach operations.
- **ALL**: Cascade all entity state change operations (persist, merge, remove, refresh, detach) from source to target.
- Multiple options can be specified in a comma-separated list in braces:
- `@OneToOne(cascade = {CascadeType.MERGE, CascadeType.PERSIST})`



# Java Persistence Query Language (JPQL)

- JPQL is JPA query language for querying the database
- The benefit is that it is converted into database specific dialect, thus migration between databases is usually easier then in the case of pure JDBC (avoid using nativeQuery as much as possible)

```
SELECT p FROM Person p;
```

```
SELECT p FROM Person p WHERE p.salary > 2000;
```

```
SELECT COUNT(p) FROM Person p;
```



# JOIN Between Entities

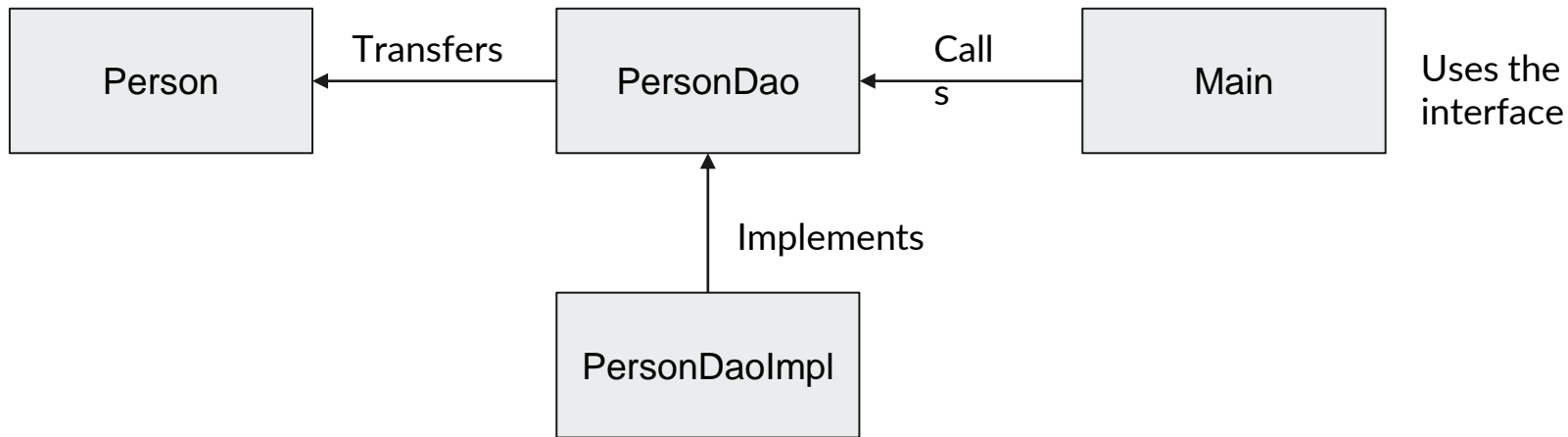
- In JPA it is essential to know JOIN FETCH
- If you want to load related entity data please write JOIN FETCH queries:

```
SELECT p FROM Person p JOIN FETCH Address
```

- This will significantly reduce the number of queries generated by JPA implementation framework



# How We Design JPA Applications?



Here we inject EntityManager and call queries on it



## And Finally... Spring Data !

- Spring Data is a wrapper over JPA
- No More DAO Implementations necessary !!
- Spring Data takes this simplification one step forward and **makes it possible to remove the DAO implementations entirely**
- The interface of the DAO is now the only artifact that we need to explicitly define
- In Spring Data it is called Repository



# Spring Data - Repository

- This is the only thing you need to configure to have basic CRUD operations on Person entity available...

@Repository

```
public interface PersonRepository extends JpaRepository<Person, Long> {  
    Person findByName(String name);  
}
```

- Based on this interface the implementation class is generated



# Spring Data - Automatic Custom Query

- The query is generated by method name

```
@Repository
public interface PersonRepository extends JpaRepository<Person, Long> {
    Person findByName(String name);
}
```



# Spring Data - Manual Custom Query - Using Parameters

```
@Repository
public interface PersonRepository extends JpaRepository<Person, Long> {
    @Query("SELECT p FROM Person p WHERE p.name = ?1")
    Person findByName(String name);
}
```



# Spring Data - Manual Custom Query

- The query is generated by method name

```
@Repository
public interface PersonRepository extends JpaRepository<Person, Long> {
    @Query("SELECT p FROM Person p WHERE p.name = :name")
    Person findByName(@Param("name") String name);
}
```




# Spring Data - DELETE/UPDATE

- DELETE or UPDATE queries have to configure also @Modifying annotation

@Modifying

@Query("DELETE FROM Person p WHERE p.id = :personId")

void deletePersonById(@Param("personId") Long personId);




# Spring Data - Pagination

- Once we have our repository **extending** from *PagingAndSortingRepository*, we just need to:
  - Create or obtain a *PageRequest* object, which is an implementation of the *Pageable* interface
  - Pass the *PageRequest* object as an argument to the repository method we intend to use
- We can create a *PageRequest* object by passing in the requested page number and the page size. Here, **the page counts starts at zero**:

```
Pageable firstPageWithTwoElements = PageRequest.of(0, 2);  
Pageable secondPageWithFiveElements = PageRequest.of(1, 5);
```
- The example in Spring Data repository:

```
Page<Person> findAll(Pageable pageable);
```






# Spring Data - Pagination

- Specifying own query with Pageable method argument..
- In that case, it is necessary to define countQuery:

```
@Query(value = "SELECT p FROM Person p WHERE p.id = :personId",  
        countQuery = "SELECT COUNT(p) FROM Person p WHERE p.id = :personId")  
Page<Person> findPersonById(@Param("personId") Long personId, Pageable pageable);
```



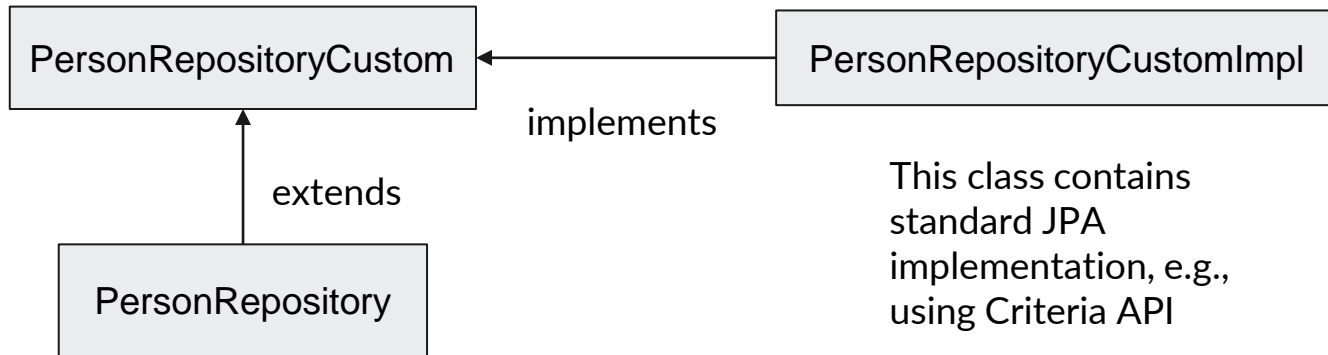
# Spring Data - Pagination

- Specifying own query with Pageable method argument..
- In that case, it is necessary to define countQuery:

```
@Query(value = "SELECT p FROM Person p WHERE p.id = :personId",  
        countQuery = "SELECT COUNT(p) FROM Person p WHERE p.id = :personId")  
Page<Person> findPersonById(@Param("personId") Long personId, Pageable pageable);
```

# Spring Data - Dynamic Queries

- Not everything could be defined in @Query
- Sometimes we need to write standard queries using EntityManager instance





# Spring Data - EntityGraph

- **JPA 2.1 has introduced the Entity Graph feature as a more sophisticated method of dealing with performance loading.**
- It allows defining a template by grouping the related persistence fields which we want to retrieve and lets us choose the graph type at runtime.

```
@EntityGraph(attributePaths = {"address", ""})  
Optional<Person> findById(Long id);
```

- <https://www.baeldung.com/jpa-entity-graph>



# Spring Data - Named Queries

- A named query is declared on an entity by using the `@NamedQuery` annotation

```
@Entity
```

```
@NamedQuery(name="Person.findByEmail",
```

```
    query="SELECT p FROM Person p WHERE p.email = :email")
```

```
public class Person {
```

```
    //...
```

```
}
```

- A named query is created with the `createNamedQuery` method



# Query Best Practices

- Use named queries whenever possible
  - Named queries are compiled and highly optimized by providers
  - Dynamic queries will always be less efficient than named queries
- A named query is created with the `createNamedQuery` method
- Always load to `PersistenceContext` all the data you need in transaction in as less queries as possible



# Spring Data - Configuration

- The only necessary configuration is configuring `@EnableJpaRepositories`

```
@EnableJpaRepositories(basePackages = "org.sedaq.persistence.repository")
public class PersistenceConfig {
    ...
}
```

- In property file:  
spring.datasource.url=jdbc:h2:mem:db;DB\_CLOSE\_DELAY=-1  
spring.datasource.username=sa  
spring.datasource.password=sa



## JPA - What We Did Not Cover

- Advanced Modelling (Inheritance, ...)
- @ElementCollection, @Embeddable, Map
- Criteria API
- Advanced JPA features
- StoredProcedures
- Caching
- Indexes





## **Section: Developing Service Layer**



## Section: Developing Service Layer

- a. Reviewing problem statement
- b. Consuming another REST service (synchronously vs asynchronously)
  - i. RestTemplate
  - ii. WebResource
  - iii. Declarative REST Feign Client
- c. Consuming another SOAP service
  - i. Implementing SOAP Clients (wsimport)
  - ii. Dispatcher Client

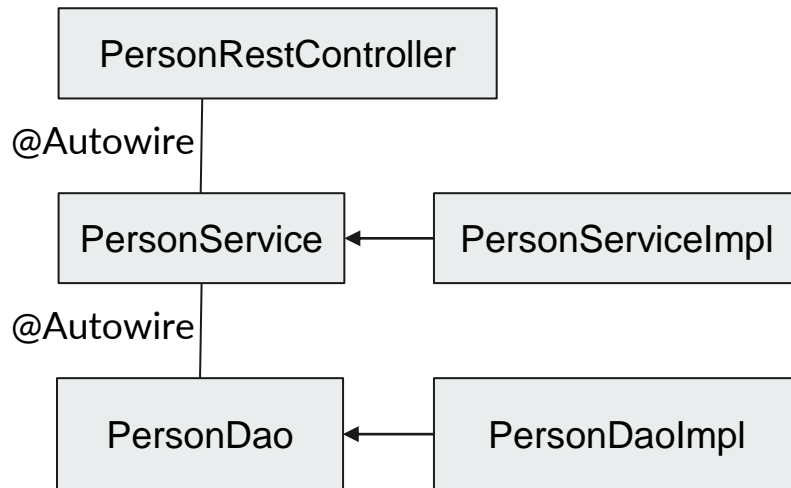


# Developing Service Layer

- Service layer is used for developing business logic
- It contains mapping from entity classes to DTO classes
- It contains calls to another microservices
- It demarcates transactions
- Authorization checks

# Developing Service Layer


- N-tier architecture





# Developing Service Layer

Service layer classes should be annotated with @Service annotation



```
@Service
@Transactional
public class PersonService {
    private PersonRepository personRepository;

    @Autowired
    public PersonService(PersonRepository personRepository) {
        this.personRepository = personRepository;
    }

    public Optional<Person> findById(Long id) {
        try {
            return personRepository.findById(id);
        } catch (HibernateException ex) {
            throw new ServiceLayerException(ex);
        }
    }
}
```

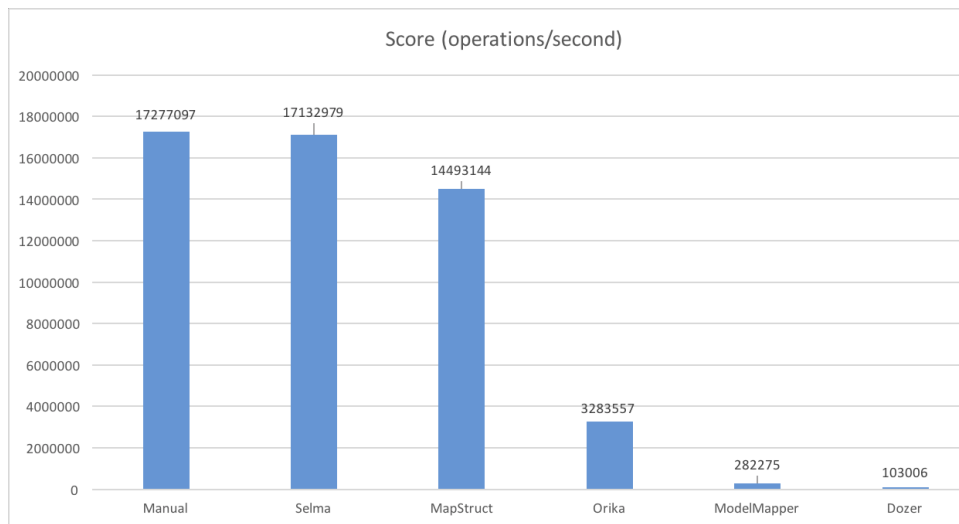


# What is DTO Class? Why Do We Need It?

- A Data Transfer Object (DTO) is an object that is used to encapsulate data, and send it from one subsystem of an application to another.
- DTOs are most commonly used by the Services layer in an N-Tier application to transfer data between itself and the UI layer. The main benefit here is that it reduces the amount of data that needs to be sent across the wire in distributed applications. They also make great models in the MVC pattern
- How to map entities to DTO classes?
  - Manually? -> the best performance but too difficult to implement..
  - Libraries? Yes!
  - In the past, mappers as Dozer or ModelMapper were used, these uses reflection, it is slow, but easy to implement
  - Nowadays mappers as MapStruct or Selma are popular, these generated some-like manual implementation

# Mappers Comparison

- Great article about mappers performance comparison is like always on baeldung..
- <https://www.baeldung.com/java-performance-mapping-frameworks>





# Consuming Another REST Service

- RestTemplate

```
HttpEntity<String> entity = new HttpEntity<>(createHttpHeaders("user", "userPwd"));
ResponseEntity<PersonDTO> response = restTemplate
    .exchange(REST_URI + "/" + id, HttpMethod.GET, entity, PersonDTO.class);
if (response.getStatusCode().isError()) {
    // log user not found
}
return response.getBody();
```

- RestTemplate has several useful methods:

- getForObject
- postForObject





# Service Layer - Transactions

- Container-Managed Transactions are placed above methods to demarcate the transaction scope
- ACID
  - Atomicity - each transaction is treated as a single “unit”
  - Consistency - ensures that a transaction can only bring the database from one valid state to another
  - Isolation - ensures that concurrent execution of transactions leaves the database in the same state that would have been obtained if the transactions were executed sequentially
  - Durability - once a transaction has been committed, it will remain committed even in the case of a system failure
- Transactions usually uses pessimistic locking (database table or row is locked until the transaction finishes the execution)



# Transactions - Container's Behaviour

Transactions propagation:

- **REQUIRED:**
  - The method becomes part of the caller's transaction. If the caller does not have a transaction, the method runs in its own transaction
- **REQUIRES\_NEW:**
  - The method always runs in its own transaction. Any existing transaction is suspended
- **NOT\_SUPPORTED:**
  - The method never runs in a transaction. Any existing transaction is suspended
- **SUPPORTS:**
  - The method becomes part of the caller's transaction if there is one. If the caller does not have a transaction, the method does not run in a transaction
- **MANDATORY:**
  - It is an error to call this method outside of a transaction
- **NEVER**
  - It is an error to call this method in a transaction



# Transactions Scopes and Entity Synchronization

- Entity components must be synchronized with the underlying database at least once per transaction.
- Entity classes do not declare transactions. They inherit the transaction of the calling component.
- Entity classes can use optimistic locking. When a transaction commits entity data, the transaction can fail to commit because the data was modified by a concurrently running transaction



# Own Annotations - For Read Only Operations

```
@Transactional(rollbackFor = Exception.class, readOnly = true)
@Target({ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Inherited
@Documented
public @interface TransactionalRO {
    @AliasFor("transactionManager")
    String value() default "";
    @AliasFor("value")
    String transactionManager() default "";
    Propagation propagation() default Propagation.REQUIRED;
    Isolation isolation() default Isolation.DEFAULT;
    int timeout() default -1;
}
```



# Own Annotations - For Write Only Operations

```
@Transactional(rollbackFor = Exception.class)
@Target({ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Inherited
@Documented
public @interface TransactionalWO {
    @AliasFor("transactionManager")
    String value() default "";
    @AliasFor("value")
    String transactionManager() default "";
    Propagation propagation() default Propagation.REQUIRED;
    Isolation isolation() default Isolation.DEFAULT;
    int timeout() default -1;
}
```



## **Section: Spring Model View Controller (MVC)**



# Spring Model View Controller (MVC)

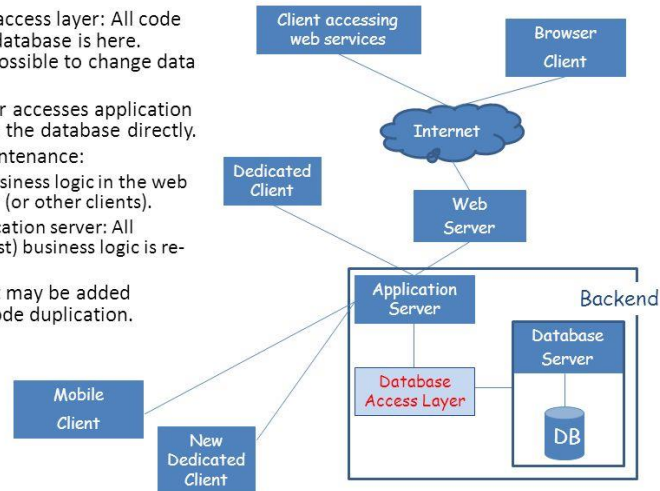
## Spring Model View Controller (MVC)

- a. Servlets
  - i. Dispatcher Servlet
- b. Java Server Pages (JSP), Java Server Faces (JSF)
- c. MVC Design Pattern
- d. Common annotations:
  - i. @Controller, @ModelAttribute, @RequestMapping, @SessionAttributes
- e. Validation (javax.validation package)
- f. Embedded application servers (Tomcat, Jetty)

# Layers in Multi-tier Application

## N-tier (multi-tier) Architecture

- Database access layer: All code to access database is here. Makes it possible to change data store.
- Web server accesses application layer – not the database directly.
- Easier maintenance:
  - No business logic in the web server (or other clients).
  - Application server: All (almost) business logic is re-used.
- New client may be added without code duplication.







# SaaS Cloud

- web applications are Software-as-a-Service type of cloud service
- provide on-demand access to software
- device independence – PC, notebook, tablet,
- smartphone, smart TV, ...
- web mail, messaging, office suites, media libraries,
- communication tools, business sw ...
- Gmail, Facebook, Google Drive, Dropbox, Spotify,
- Flickr, YouTube, WebEx, NetSuite ...



# Deployment

- SaaS services can be deployed
  - Into Platform-as-a-Service (PaaS) cloud
    - Microsoft Azure, Amazon Elastic Beanstalk, IBM Cloud, RedHat OpenShift, Heroku, ...
  - into Infrastructure-as-a-Service (IaaS) cloud
    - Google Computing Engine, Amazon Elastic Compute Cloud, Microsoft Azure, ...
  - Locally
- software is provided as
  - downloadable executable code (i.e. JavaScript, Android app)
  - callable API on provider's servers (e.g. Google Calendar API)



# Client Side Technologies

- HTML, forms, CSS
  - Cookies
  - JavaScript, Document Object Model, AJAX
  - HTML5 features - <canvas>, <video>, web storage, web sockets, file API, geolocation API, device orientation, media capture
  - Scalable Vector Graphics (SVG)
- 
- Dead technologies: Java applets, Flash
  - Soon will be dead: JSP, JSF

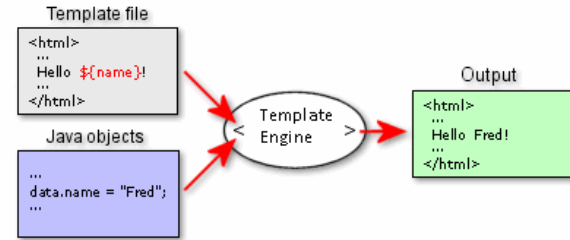


# Server Side Technologies

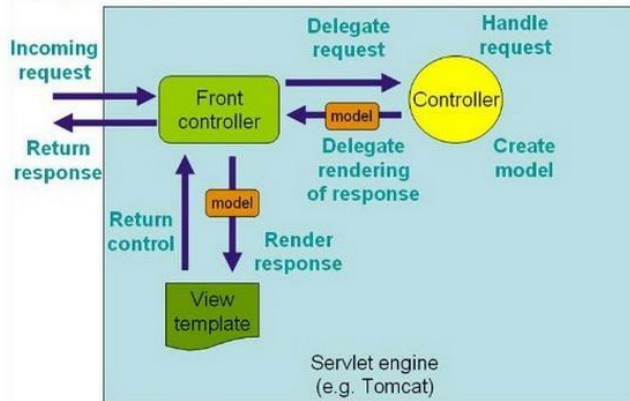
- Python, C#, PHP, Ruby on Rails, NodeJS, ...
- Java web containers (Java EE implementations) - Apache Tomcat, Jetty, JBoss, IBM WebSphere, ...)
- Servlet API for handling HTTP
- Java Server Pages (JSP) for page templates
- Frameworks on top of Servlet API

# Java Server Side Frameworks

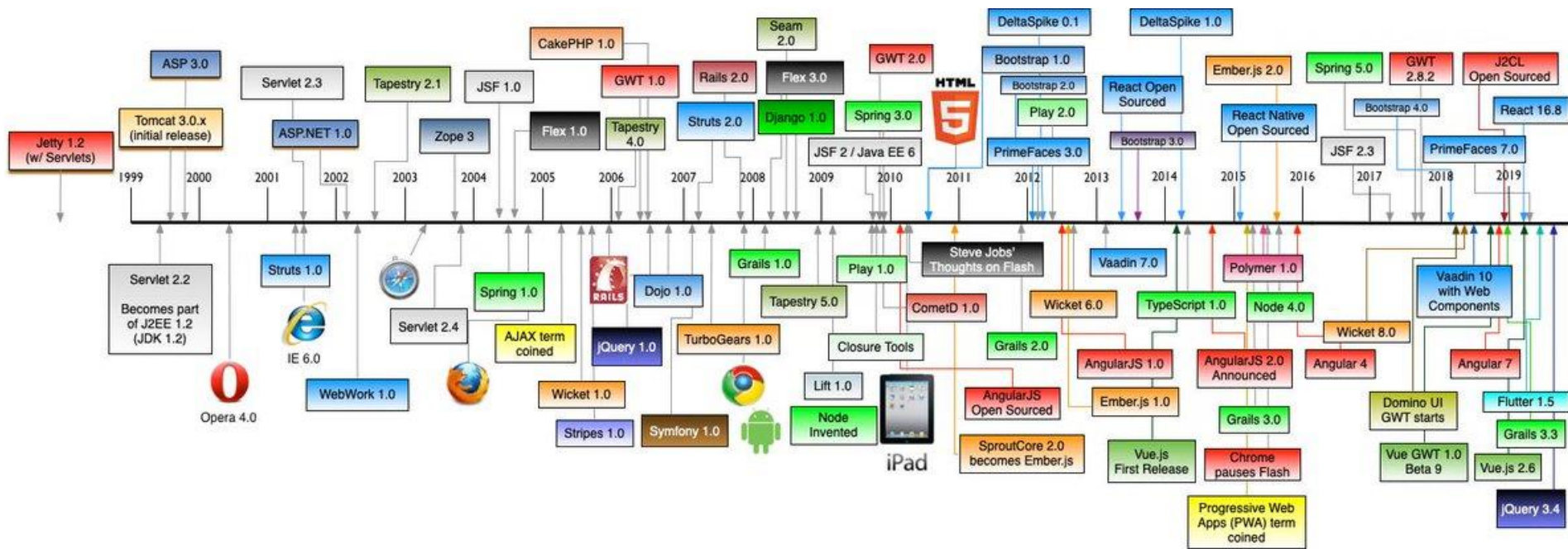
- Page templates
  - JSP, tag libraries, JSTL
  - Velocity
  - Freemaker
- Model-View-Controller
  - Spring MVC
  - Stripes
  - Apache Struts



## Spring MVC Workflow



# Web Frameworks History





# Servlet API

- **Servlets**: for managing HTTP requests
- **Filters**: for modifying requests and responses
- **Listeners**: for handling events (e.g., app start)
- **HttpSession**: for maintaining state
- **RequestDispatcher** and **attributes** for cooperation among multiple servlets



# Servlet API

```
@WebServlet("/persons/*")
public class PersonServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        request.getRequestDispatcher("/person.jsp").forward(request, response);
    }

    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        response.sendRedirect(request.getContextPath() + "/persons");
    }
}
```





# Filter

```
@WebFilter("/persons")
public class PersonFilter implements Filter {
    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain filterChain)
        throws IOException, ServletException {
        doSomething(servletRequest, servletResponse);
        filterChain.doFilter(servletRequest, servletResponse);
    }

    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
    }

    @Override
    public void destroy() {
    }
}
```



# Listeners

- ServletContextListener, ServletRequestListener, HttpSessionListener, ...

```
@WebListener
public class PersonContextListener implements ServletContextListener {

    @Override
    public void contextInitialized(ServletContextEvent sce) {
        ServletContext ctx = sce.getServletContext();
        ctx.setAttribute("common_object", getCommonObject());
    }

    @Override
    public void contextDestroyed(ServletContextEvent sce) {

    }
}
```



# HttpSession

- Keeps a Map<String, Object> on server
- Assigned to a particular browser
- Maintained using cookie or URL rewriting
- Timeout after 30 minutes since last request

```
@WebServlet("/auth/*")
public class ServletAuthenticationChecker extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        HttpSession session = request.getSession(true);
        Boolean authenticated = (Boolean) session.getAttribute("authenticated");
        if (!authenticated) { response.sendError(HttpServletResponse.SC_UNAUTHORIZED); }
    }
    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException { }
}
```



## WAR - Web Archive

- A servlet container can have multiple web applications running, mapped by different context path (first part of URL)
- Each web app were usually deployed in a \*.war file
- It is a ZIP archive containing:
  - WEB-INF/classes/\*\*/\*.class -> classes
  - WEB-INF/lib/\*.jar -> libraries
  - WEB-INF/web.xml -> deployment descriptor
  - WEB-INF/tags/\*.tag -> custom JSP tags
  - Directly accessible files like \*.jsp, \*.png, \*.css, \*.js



# Java Server Pages

- An HTML file with special directives, converted to a servlet on each change
- Scriptlets, EL language, tags, tag libraries

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags"%>
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<html>
    <%= pageContext.findAttribute("common_object") %>
    <% out.print(pageContext.findAttribute("a")); %>
    <c:out value="${a}" escapeXml="false"/>
</body>
</html>
```



# JSP Expression Language (EL)

- Expressions inside of `${}`
- Value expressions:
  - `${attribute.property}`
  - `${attribute['property']}`
- Operators: `+`, `-`, `*`, `/`, `div`, `%`, `mod`, `and`, `or`, `not`, `==`, `!=`, `<`, `>`, `<=`, `>=`, `empty`
- Functions defined by tag libraries
  - `${fn:length(orderitems)}`



# Own JSP Tags

- Defined in \*.tag files with syntax similar to \*.jsp
- Can be used for common page layout
- Attributes can be: simple, dynamic, fragment
  - a.tag example:

```
<%@ tag pageEncoding="utf-8" trimDirectiveWhitespaces="true" dynamic-attributes="attr" %>
<%@ attribute name="href" required="true" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<c:url value='${href}' var="url" scope="page"/>
<a href="
```



# Common Page Layout Using \*.tag

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib tagdir="/WEB-INF/tags" prefix="my"%>
<%@ taglib prefix="sec"
      uri="http://www.springframework.org/security/tags"%>
<my:pagetemplate title="Language School students">
  <jsp:attribute name="body">
    // some table showing persons
  </jsp:attribute>
</my:pagetemplate>
```





# Java Standard Tag Library (JSTL)

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>

<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

<%@ taglib prefix="spring" uri="http://www.springframework.org/tags"%>

<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form"%>


<c:set var="a" value="123" scope="request"/>
<c:out value="${a}"/>
<c:if test="${a>1}"> a is bigger than 1 </c:if>
<c:forEach items="${cars}" var="car" varStatus="i"> ${i.count}: ${car.id} </c:forEach>
<c:choose>
    <c:when test="${a<0}"> a is negative </c:when>
    <c:when test="${a==0}"> a is zero</c:when>
    <c:otherwise> a is positive</c:otherwise>
</c:choose>
```



# The Problems of Today's Web Design

- Wide range of screen sizes from 3" phones to 30" desktop monitors
- Wide range of pixel densities (80 ppi - 560 ppi)
- Touch screens do not have "mouse over" events
- Devices change orientation (portrait / landscape)



# Responsive Web Design

- Web design that adapts to screen size and pixel density
- CSS media queries
  - @media screen and (min-width: 400px) {...}
- CSS pixels versus hardware pixels
  - CSS pixels are 96ppi at 28" distance (1px=0.26mm)
  - Hardware pixels described in CSS by device-pixel-ratio
    - device-pixel-ratio: 2 – iPhone4, iPad3
    - device-pixel-ratio: 3 - Galaxy S4, LG G3, HTC One
    - device-pixel-ratio: 4 - Galaxy Note Edge, Xiaomi Mi3
- Images should be served in HW pixel resolutions

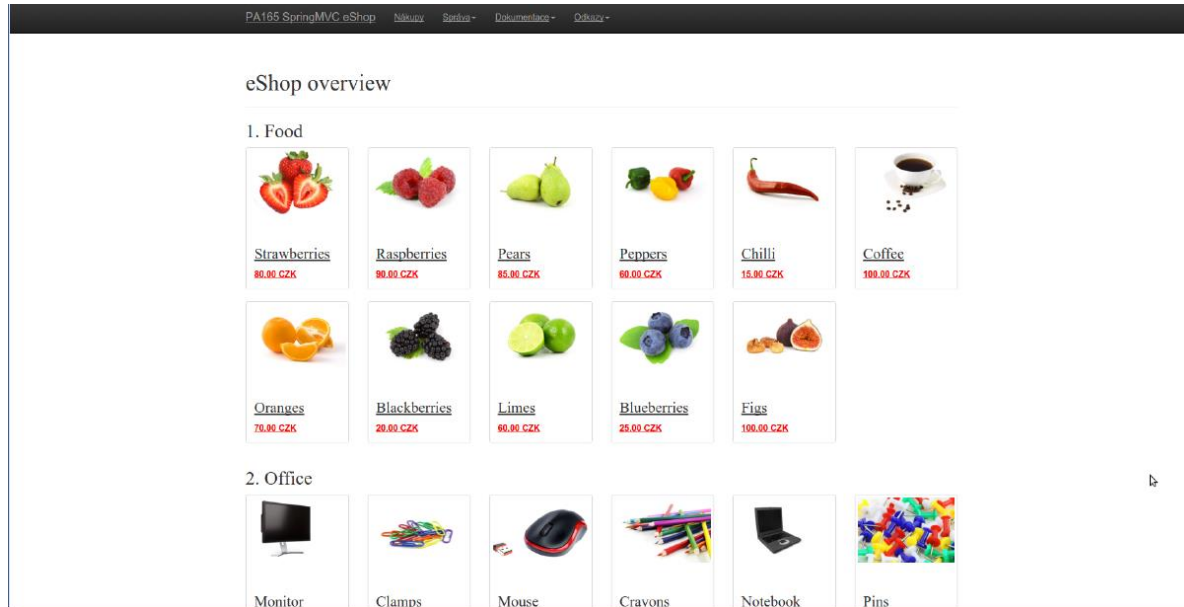


# Bootstrap

- Originally created by Twitter
- It is CSS framework for responsive web design (nowadays, the de-facto standard)
- Navigation menu collapses on small screens
- 12-column grid for positioning
- 4 screen sizes: extra small, small, medium, large
- CSS classes for rows and columns

```
<div class="row">  
  <div class="col-xs-6 col-sm-8 col-md-9 col-lg-10"></div>  
  <div class="col-xs-6 col-sm-4 col-md-3 col-lg-2">  
    <div class="panel panel-default".../>  
  </div>  
</div>
```

# Desktop 24" 1920x1080 90ppi



# Tablet 10" 1920x1200 224ppi

PA165 SpringMVC eShop

Nákupy

Správa ▾

Dokumentace ▾

Odkazy ▾

## eShop overview

### 1. Food



Strawberries

80.00 CZK



Raspberries

90.00 CZK



Pears

85.00 CZK



Peppers

60.00 CZK



Chilli

15.00 CZK



Coffee

100.00 CZK



Oranges

70.00 CZK



Blackberries

20.00 CZK



Limes

60.00 CZK



Blueberries

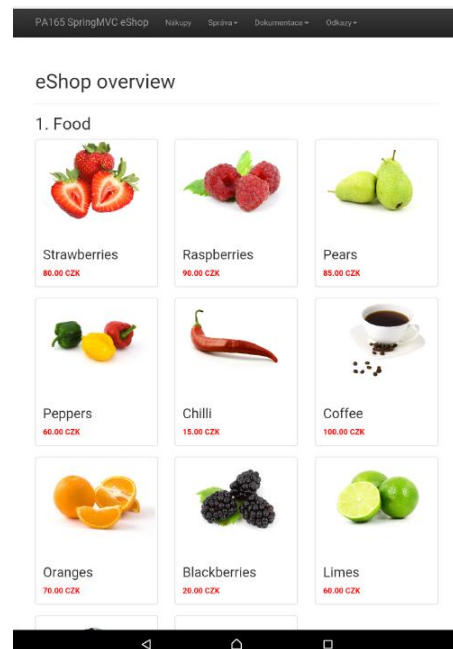
25.00 CZK



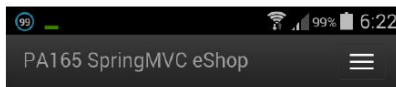
Figs

100.00 CZK

# The same 10" in Portrait Mode



4.3" 540x960 256 ppi



## eShop overview

### 1. Food







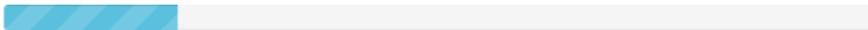
# Bootstrap Additional Features

- Vector icons
- Support for screen readers
- Drop-down menus
- Buttons and button groups
- Badges
- Alerts
- Progress bars



**Well done!** You successfully read this important alert message.

**Heads up!** This alert needs your attention, but it's not super important.





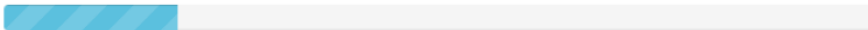
# Bootstrap Additional Features

- Vector icons
- Support for screen readers
- Drop-down menus
- Buttons and button groups
- Badges
- Alerts
- Progress bars



**Well done!** You successfully read this important alert message.

**Heads up!** This alert needs your attention, but it's not super important.





# Material Design

- Developed in 2014 by Google, **Material Design** is a “design language”. It is based on the “card” motifs that were first introduced in Google Now, and has expanded on that by including responsive transitions and animations, effects such as lighting and shadows and grid-based layouts
- Being a design language, Material Design defines a set of guidelines, which showcase how to best design a website. It tells you what buttons are for use and which ones you should use, how to animate or move it, as well as where and how it should be placed, etc.
- Providers:
  - Google material design: <https://material.io/design/>
  - IBM material design: <https://www.carbondesignsystem.com/>



# Material Design vs Bootstrap

- **Philosophy and Purpose**

- While both of them are used for the purpose of web development, **Material Design** is more **oriented towards the way that a website or application will look**. This is evident in the way it is used, its design guidelines and “rules” and the numerous templates and components it comes with, which are focused on design
- **Bootstrap**, on the other hand, is mainly focused on easily creating responsive websites and web applications, which are functional and of high quality when it comes to user experience

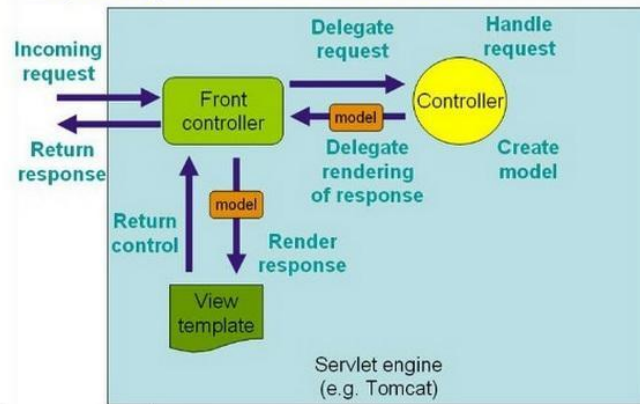
- **Design Process and Components**

- **Material Design comes with numerous components** that provide a base design, which then can be worked on by the developers themselves. It strictly follows the patterns of Material Design concept, which provides the necessary information on how to use each component
- Bootstrap is an open-source framework, also referred to as a UI library, which can use **Grunt** to create and run its designing processes

# Spring MVC

- One of many Spring libraries, optional
- Model View Controller architecture
- Request-driven framework

## Spring MVC Workflow





# Spring MVC Initialization

- By hand:
  - Initialize a new `DispatcherServlet` instance with `WebApplicationContext`
  - Add the servlet instance to your web app
- Automatically:
  - Extend `AbstractAnnotationConfigDispatcherServletInitializer` and implements its methods `getRootConfigClasses()` and `getServletMappings()`
- In both cases, provide:
  - A class annotated with `@EnableWebMvc` that configures Spring MVC
  - A class annotated with `@Configuration` that configures Spring beans
  - Can be just a single class



# Spring MVC Configuration

- A class with `@EnableWebMvc` or XML based
- Should provide:
  - `ViewResolver` for resolving views, e.g., JSPs
  - `MessageSource` for localized messages
  - `Validator` for validating data in beans
- Can enable default servlet for static files



# Spring MVC Initialization - Automatically

```
public class StartInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {  
    @Override  
    protected Filter[] getServletFilters() {  
        CharacterEncodingFilter encodingFilter = new CharacterEncodingFilter();  
        encodingFilter.setEncoding("UTF-8");  
        return new Filter[] { encodingFilter };  
    }  
    @Override  
    protected Class<?>[] getRootConfigClasses() { return new Class<?>[] { SpringMVCConfig.class };}  
    @Override  
    protected String[] getServletMappings() { return new String[] { "/" };}  
    @Override  
    protected Class<?>[] getServletConfigClasses() { return null; }  
}
```





# Spring MVC Configuration

```
@EnableWebMvc
@Configuration
@ComponentScan(basePackages = { "org.sedaq.mvc.controllers" })
public class SpringMVCConfig extends WebMvcConfigurerAdapter {

    ...

}
```



# Spring MVC Configuration - ViewResolver

```
@Bean
public ViewResolver viewResolver() {
    logger.debug("registering JSP in /WEB-INF/jsp/ as views");
    InternalResourceViewResolver viewResolver = new InternalResourceViewResolver();
    viewResolver.setContentType("text/html; charset=UTF-8");
    viewResolver.setPrefix("/WEB-INF/jsp/");
    viewResolver.setSuffix(".jsp");
    return viewResolver;
}

@Override // Maps the main page to a specific view
public void addViewControllers(ViewControllerRegistry registry) {
    logger.debug("mapping URL / to login view");
    registry.addViewController("/").setViewName("login");
}
```



# Spring MVC Configuration - MessageSource

```
@Bean
public MessageSource messageSource() {
    logger.debug("registering ResourceBundle 'Texts' for messages");
    ResourceBundleMessageSource messageSource = new ResourceBundleMessageSource();
    messageSource.setBasename(TEXTS);
    messageSource.setDefaultEncoding("UTF-8");
    return messageSource;
}
```



# Spring MVC Configuration - Validator/Static Files

```
@Bean
public Validator validator() {
    logger.debug("registering JSR-303 validator");
    return new LocalValidatorFactoryBean();
}

/**
 * Enables default Tomcat servlet that serves static files.
 */
@Override
public void configureDefaultServletHandling(DefaultServletHandlerConfigurer configurer) {
    logger.debug("enabling default servlet for static files");
    configurer.enable();
}
```



# Spring MVC Controllers

```
@Controller
@RequestMapping("/person")
public class PersonController {

    @Autowired
    private PersonService personService;

    @RequestMapping("/foo")
    public String foo(@RequestParam int a, Model model){
        // pass data as request attributes to views
        model.addAttribute("b", a+1);
        // ViewResolver resolves to /WEB-INF/jsp/foo.jsp
        return "foo";
    }
}
```



# Spring MVC Controllers

- Any class annotated with `@Controller`
- Mapping of methods to URLs is set by `@RequestMapping`, can have common prefix for the whole class
- Dependencies are injected using `@Autowired`
- Can return `String`, which is resolved by `ViewResolver` (provided by `@EnableWebMvc`) to view, usually a JSP page
- Data are passed through instance of `Model`
- Method parameters specify inputs
- Automatic type conversion for request params and path



# Spring MVC Controllers - Method Parameters

```
@RequestMapping("/foo/{a}/{r1:[a-z]+}{r2:\\d+}")
public String foo2(
    @PathVariable int a,
    @PathVariable String r1,
    @RequestParam("b") long b,
    Locale locale,
    HttpMethod httpMethod,
    @RequestHeader("User-agent") String userAgent,
    @CookieValue("mycookie") Cookie mycookie,
    Model model,
    HttpServletRequest req,
    HttpServletResponse res
) {
    return "foo";
}
```



# Spring MVC Controllers - Redirect

- Triggered by return value starting with “redir:”

```
@RequestMapping("/redir")
public String someRedirect(Locale locale, RedirectAttributes redirectAttributes){
    redirectAttributes.addAttribute("pid", 21);
    redirectAttributes.addAttribute("cid", 33);
    return "redirect:/product/{pid}/category/{cid}";
}
```

- RedirectAttributes
  - Attributes replace placeholders {attr} in URL
  - @PathVariable parameters automatically added as attributes
  - Provide so called flash attributes, which exist only during the first next request
- More complex URL building possible using UriComponentsBuilder class
- Redirect after POST! To avoid duplicate submissions





# Spring MVC - Order Controller

```
@Controller
@RequestMapping("/order")
public class OrderController {
    @Autowired
    private OrderFacade orderFacade;

    @RequestMapping(value = "/view/{id}", method = RequestMethod.GET)
    public String view(@PathVariable long id, Model model) {
        model.addAttribute("order", orderFacade.getOrderWithId(id));
        return "order/view";
    }

    @RequestMapping(value =("/{id}", method = RequestMethod.DELETE)
    public String removeOrder(@PathVariable long id,
        RedirectAttributes redirectAttributes) {
        orderFacade.deleteOrder(id);
        redirectAttributes.addFlashAttribute("message", "Order was deleted.");
        return "redirect:/order/list";
    }
}
```



# Spring MVC - Tag Library For Forms

- Binds form fields to bean properties
- Display error messages when validation fails
- Keeps values entered by user when validation fails

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags"%>
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form"%>

<form:form method="post" action="/order" modelAttribute="orderCreate">

    <form:label path="name">Name</form:label>
    <form:input path="name" />
    <form:errors path="name"/>
    <button type="submit">Create</button>
</form:form>
```



# Spring MVC - @Valid and BindingResults

- The form is passed to the method as Java Bean class

```
@RequestMapping(method = RequestMethod.POST)
public String createOrder(@Valid @ModelAttribute OrderCreateDto orderCreateDto,
                          BindingResult bindingResult,
                          Model model,
                          RedirectAttributes redirectAttributes) {
    if (bindingResult.hasErrors()) {
        for (FieldError fe : bindingResult.getFieldErrors()) {
            model.addAttribute(fe.getField() + "_error", true);
        }
        return "product/new";
    }
    //create order
    Long id = orderFacade.createOrder(orderCreateDto);
    redirectAttributes.addFlashAttribute("alert_success", "Order was created.");
    redirectAttributes.addAttribute("id", id);
    return "redirect:/product/view/{id}";
}
```



# Spring MVC - Input Data Validation

- JSR-303 “Bean Validation” provides annotations and validators for java bean properties
- Hibernate Validator is implementation of JSR-303
- @NotNull, @Max, @Min, @Size, @Future, @Past, @Pattern, ...
- A single definition of validation reused in various layers - e.g., persistence and web forms
- You can define your own annotations and provide its validator and localized error messages
- Class with @EnableWebMvc has to provide Validator instance (in Spring Boot provided automatically :))



# Spring MVC - Input Data Validation

```
public class OrderCreateDto {  
  
    @NotNull  
    @Size(min = 3, max = 50)  
    private String name;  
    @NotEmpty  
    @Size(min = 3, max = 500)  
    private String description;  
    @NotNull  
    @Min(0)  
    private BigDecimal price;  
    @Email(message = "Email must be provided.")  
    private String contactEmail;  
    @Pattern(regexp = "\\d+")  
    private Long pin;  
}
```



# Spring MVC - Summary

- Spring MVC uses controllers to process HTTP requests (it is based on Java EE Servlets API)
  - Send Model to views to display
  - Or send redirects (always after POST)
- Flash attributes for passing data through redirects
- Form tag library helps in form handling
- Request parameters may be bound to properties of a method parameter with `@ModelAttribute`
- JSR-303 Bean Validation is commonly used for input object validation



## **Section: Spring REST Services**



# Section: Spring REST Services

## Spring REST Services

- a. Rest architecture
- b. Common annotations:
  - i. `@RestController`, `@PathVariable`, `@RequestParam`, `@RequestBody`
- c. Data Transfer Object (DTO) classes
- d. Jackson
- e. Exception handling (`@RestControllerAdvice`)
- f. Swagger Documentation
- g. Testing Tools
  - i. Postman, SoapUI, Browser plugins





# REST Architecture

- Representational State Transfer (REST) is architectural style for distributed hypermedia systems
  - It is architecture! Not protocol, that is the difference between REST and SOAP
  - REST was defined in 2001 in Roy Fielding's doctoral dissertation thesis
    - [http://www.ics.uci.edu/~fielding/pubs/dissertation/rest\\_arch\\_style.htm](http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm)
  - The original idea behind Representational State Transfer is to mimic the behaviour of Web applications : as a net of Web pages and links, resulting in the next page(state change)
  - REST was born in the context of HTTP, but it is not limited to that protocol



# REST Constraints

1. **Uniform interface:** You **MUST** decide APIs interface for resources inside the system which are exposed to API consumers and follow religiously
2. **Client-server:** This essentially means that client application
3. **Stateless:** Inspired by HTTP, server will not store anything about latest HTTP request cl. made
4. **Cacheable:** In REST, caching shall be applied to resources when applicable
5. **Layered system:** A client communicates only with an adjacent layer. A Service should not expose any implementation details
6. **Code on demand** (optional): Most of the time you will return resources in JSON (or in the past, XML) form, but when you need you are free to return, e.g., UI widget, or .ZIP or whatever you will need



# REST Resources

- The key abstraction of information in REST is a **resource**
  - A resource identification is through URI
  - Resources consist of data and metadata (metadata are often in the form of HTTP headers (eTag, etc.))
  - A RESTful web service is designed by identifying the resources
- In hypermedia-based REST, the resources are interconnected by hyperlinks



# REST Design

- The key abstraction of information in REST is a **resource**
  - A resource identification is through URI
  - Resource consists of data and metadata (metadata are often in the form of HTTP headers (eTag, etc.))
  - A RESTful web service is designed by identifying the resources
- In hypermedia-based REST, the resources are interconnected by hyperlinks

https : //localhost : 8080 / sedaq-ar-rest / api/v1 / games/25/customers  
protocol                      host                      port                      project name                      versioning                      REST resource



## REST Action Resources

- Sometimes, it is required to expose an operation in the API that inherently is non RESTful
- Consider the following situation:
  - <http://localhost/my-rest/api/v1/accounts/1>
  - <http://localhost/my-rest/api/v1/accounts/2>
  - We want to transfer money between account 1 and account 2 ...
- What to do?
  - The verb we need to do is transfer, so by this verb we could identify the needed action resource:
  - <http://localhost/my-rest/api/v1/transfers>



# REST Data Input

- URL path parameter: <https://localhost/sedaq-ar-rest/api/v1/games/{id}>
  - Use Path variable typically for retrieving specific record by id from a collection of resources
  - Do not use <https://localhost/sedaq-ar-rest/api/v1/games?id=1>
- Query parameters:
  - Use query parameters to filter, sort, or limit collections
  - <https://localhost/sedaq-ar-rest/api/v1/games?name=pavel&email=seda@email.cz>
- Headers:
  - Send metadata or links through HTTP headers (Content-Type, Accept, Authentication, ...)



# REST Methods

Method	Collection of resources	Single
Item		
GET	Get a list of all the resources	
	Retrieve data for resource with id 1	
PUT	Update the collection with a new one	Update the
resource with id 1		
POST	Create a new member resource	
	Create a sub-resource	
DELETE	Delete the whole collection	Delete the
resource with id 1		
HEAD	Retrieve metadata information	Retrieve
data for resource with id 1		



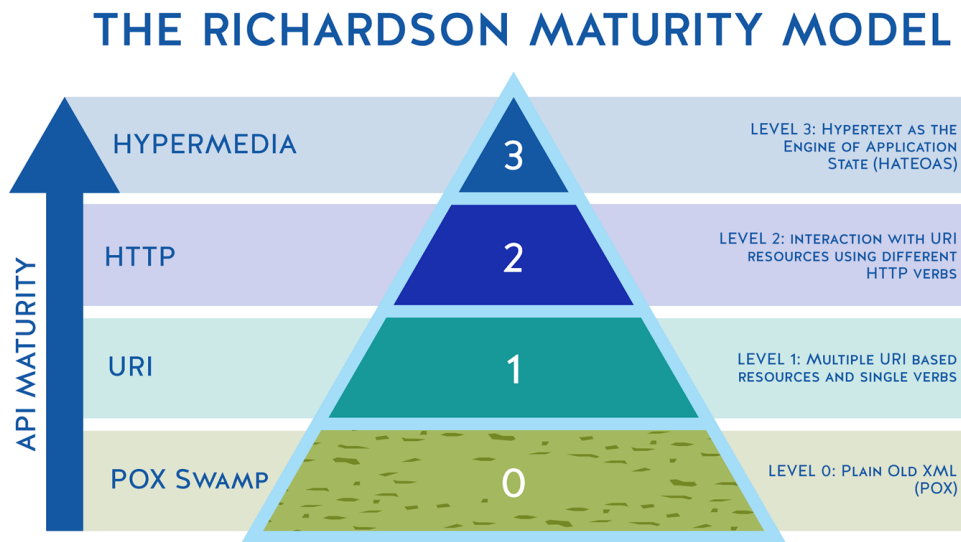
# REST HATEOAS

- Hypermedia as the Engine of Application State (HATEOAS)
- With HATEOAS, a client interacts with a network application whose application servers provide information dynamically through [hypermedia](#)
- A REST client needs little to no prior knowledge about how to interact with an application or server beyond a generic understanding of hypermedia
- Based on Roy-Fieldings dissertation REST without Hypermedia is not a REST
  - But... This was not widely accepted by industry..
  - The idea is good, you need to know only the root page, e.g., <https://gopas.cz> and then navigate through hyperlinks to other pages
  - But.. It is quite difficult to implement on server and on client side
  - It is not supported (at least I know) by Swagger documentation



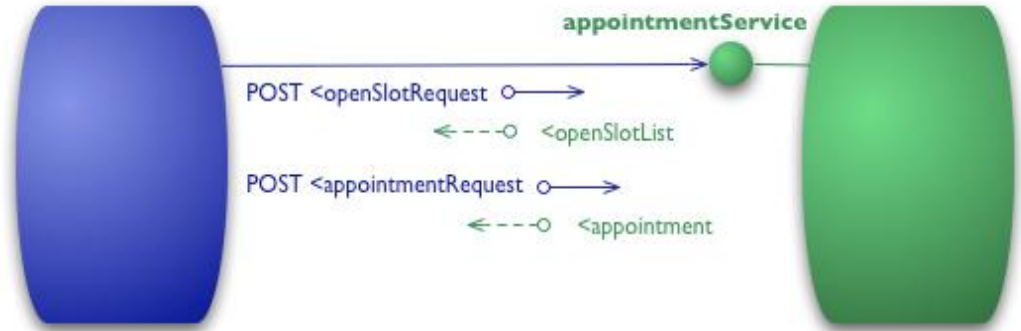
# Richardson Maturity Model

- Explains different levels at which REST can be implemented
- For further, information see:  
<https://martinfowler.com/articles/richardsonMaturityModel.html>



## Level 0 - The SWAMP of POX\*

- Looks more as Remote Procedure Call System
- We POST to and endpoint asking for different services
- There is no knowledge about resources, rather messages that are send to the endpoints (and back responses)

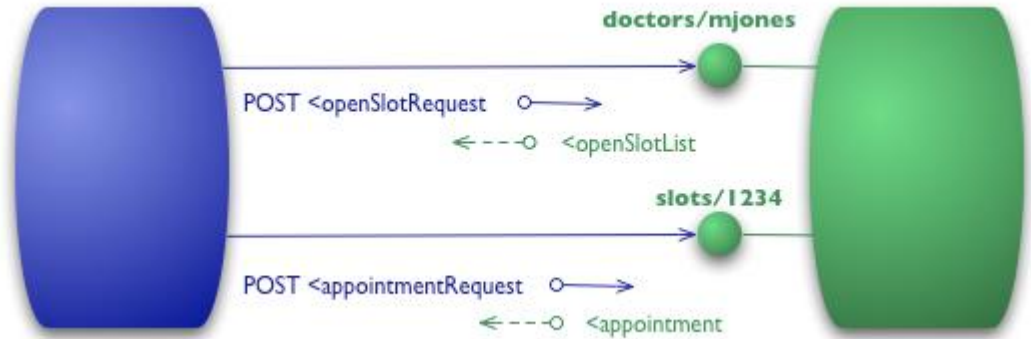


```
POST /appointmentService HTTP/1.1  
[various other headers]
```

```
<openSlotRequest date = "2010-01-04" doctor = "mjones"/>
```

## Level 1 - Resources

- At this level we introduce Resources
- We contact resources, not endpoints
- Instead of passing parameters, now we contact the specific resource

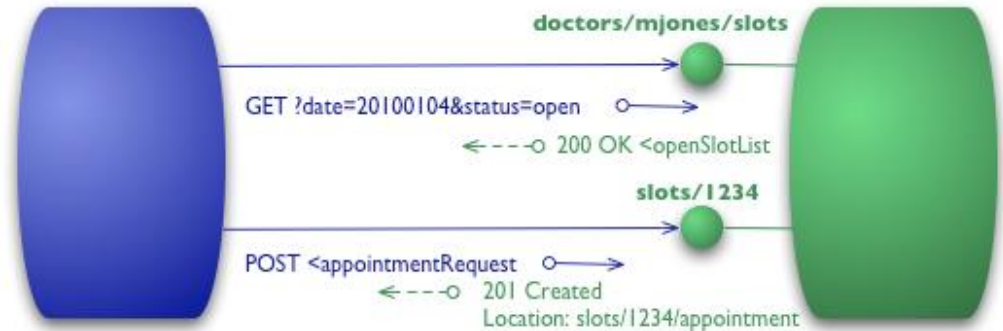


```
POST /doctors/mjones HTTP/1.1
[various other headers]
```

```
<openSlotRequest date = "2010-01-04"/>
```

## Level 2 - HTTP Verbs

- At this level we start using HTTP verbs
- We start differentiating between POST and GET
- We also start using HTTP response codes
- We start differentiating “safe” vs “unsafe” operations



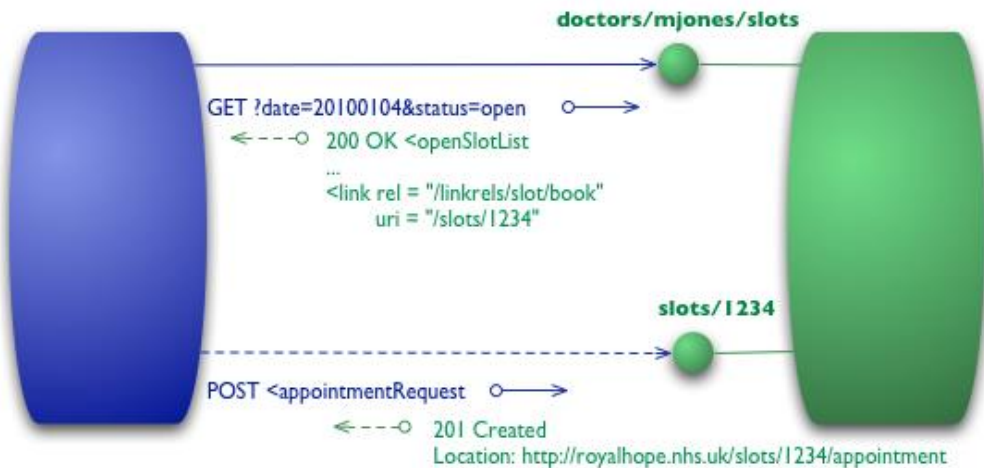
## Level 3 - Hypermedia Controls

- We introduce HATEOAS

```
GET /doctors/mjones/slots?date=20100104&status=open HTTP/1.1
Host: royalhope.nhs.uk
```

```
HTTP/1.1 200 OK
[various headers]
```

```
<openSlotList>
  <slot id = "1234" doctor = "mjones" start = "1400" end = "1450">
    <link rel = "/linkrels/slot/book"
      uri = "/slots/1234"/>
  </slot>
  <slot id = "5678" doctor = "mjones" start = "1600" end = "1650">
    <link rel = "/linkrels/slot/book"
      uri = "/slots/5678"/>
  </slot>
</openSlotList>
```





# Safety and Idempotence

- The term “**safe**” means that if a given method is called, the resource state on the server remains unchanged
- By specification, GET and HEAD should always be safe - clearly it is up to the developers not to violate this hidden specification
- PUT, DELETE are considered unsafe, while for POST generally depends
- The word “**idempotent**” means that, independently from how many times a given method is invoked, the end result is the same
- **GET** and **HEAD** are an example of an **idempotent** operation
- **PUT** is as well **idempotent**: If you add several times the same resource, it should be only inserted once
- **DELETE** is as well **idempotent**: issuing delete several times should yield the same result - the resource is gone (but what about DELETE /items/last ?)
- **POST** is generally **not** considered an **idempotent** operation



# Safety and Idempotence

Method		Safety	Idempotence
GET			YES
		YES	
PUT			NO
		YES	
POST			NO
		NO	
DELETE		NO	
	YES		
HEAD		YES	
	YES		



## REST - versioning

- When to create version 2? api/v2? Retire version 1?
- Private internal services are easier to retire, public services less so
- No breaking changes: If possible, enhance your existing service by adding additional functionality (new URL or HTTP methods) but do not change anything that is in use by clients
- Parallel versions: Each version has a different root resource URL. /v1 and /v2/ are common
- Keep the version 1 URLs but use custom content types to switch to different behaviours

https : //localhost : 8080 / sedaq-ar-rest / api/v1 / games/25/customers  
protocol                      host                      port                      project name                      versioning                      REST resource





# REST - HTTP Status Codes

Code	Meaning	Description
200	Everything processes OK	OK
201	Resource is created	Created New
202	Accepted for processing but not yet completed	Accepted
301	Moved permanently	Requests should be redirected to the given URI
400	syntax or the request cannot be fulfilled	Bad request Bad
401		Unauthorized



## REST - Best Practices (1/3)

1. Have consistent usage of **resource names**, e.g., plural for resources -> **/users/1, orders/1**
2. **Use nouns not verbs in URIs**: Each resource is following structure as follows:

Resource	GET	POST	PUT	DELETE
~/games	Returns the list of games	Create a new game	Mass actualization of games	Delete all games
~/games/1	Return game with id 1	Unsupported method	Actualization of a specific game	Delete specific game

This structure shows that the REST should not contains resources as **~/getAllGames**



## REST - Best Practices (2/3)

3. **HTTP header for format serialization:** Message format must be specified in HTTP header. **Content-Type** in HTTP header defines the request format and **Accept** defines the list of accepted response formats
4. **Allow possibility to filter, sort, selection of returned attributes and pagination:**
  - a. GET ~/games?sort=ASC
  - b. GET ~/games?type=action\_games
  - c. GET ~/games?page=3
  - d. GET ~/games?fields=[id,name,title,description]
5. **API versioning:** Create API versioning in URI with the usage of ordinal numbers only! ~/api/v1
6. **Exception handling:** Without exception handling it is almost impossible to debug errors:
7. **Use sub-resources for relations:** If resource has a relation with another resource, it is suitable to use so-called sub-resource: ~/api/v1/games/34/customers (return all customers from the game with id 34)
8. **HTTP methods PUT, POST and PATCH should return resource representation**



## REST - Best Practices (3/3)

### 9. Hyphens should be used instead of camelCase or underscore between words:

- a. CamelCase could bring problems if server does not recognize case sensitivity
- b. Underscore is bad because text searchers or editors usually underscore URIs, to provide higher visualization that this part of the text is clickable

### 10. Functions Create, Read, Update, DELETE (CRUD) should not be exposed in the URI:

### 11. Using HATEOAS:

- a. But as we discussed earlier, nowadays it is not necessary to use it

### 12. Additional best practices could be, e.g.,

- a. Using only lowercase letters in URI
- b. Not using file extension in the URI
- c. Not using symbol "/" as the last character in the URI and so on



# REST - Documentation

- Swagger:
  - <https://swagger.io/>
  - Currently, the winner of REST documentation frameworks
- Web Application Description Language (WADL):
  - Was the REST variant of **SOAP WSDL**, not used much today
- Apiary:
  - <https://apiary.io/>
- Spring REST Docs:
  - Official Spring Project: <https://spring.io/projects/spring-restdocs>
- Based on the Fieldings words, documentation is **against** the principle of REST services (in the hypermedia context)
- In a lot of companies the biggest **PAIN** for developers:
  - REST services are not much documented, or there are a lot of mistakes in it..
  - It is wrong, take a time and document everything properly!



# REST - In Spring Framework

# REST - Spring REST Controller

```
@RestController
@RequestMapping(value = "/persons")
public class PersonRestController {
```

Resource base address, on class level use  
`@RequestMapping`

path or value specify sub-resource path

```
    @GetMapping(path =("/{id}", produces = {MediaType.APPLICATION_JSON_VALUE})
    public ResponseEntity<Person> findPersonById(
        @PathVariable Long id,
        @RequestHeader HttpHeaders headers) {
        PersonDTO userResource = personFacade.findById(id);
        return new ResponseEntity<>(userResource, HttpStatus.OK);
    }
}
```

Content-Type:  
application/json

Use ResponseEntity objects




# REST - Spring REST Controller

- DELETE:

```
@RestController
@RequestMapping(value = "/persons")
public class PersonRestController {

    @DeleteMapping(path =("/{id}")
    public void deletePersonById(@PathVariable Long id) {
        personFacade.deletePerson(id);
    }
}
```



@DeleteMapping, @GetMapping,  
@PostMapping, @PutMapping  
Are available since Spring 4.3  
In the previous versions you have  
to use @RequestMapping and  
specify a HTTP method





# REST - Spring REST Controller

- UPDATE:

```
@RestController
@RequestMapping(value = "/persons")

public class PersonRestController {

    @PutMapping(path =("/{id}", produces = {MediaType.APPLICATION_JSON_VALUE},
        consumes= {MediaType.APPLICATION_JSON_VALUE})
    public ResponseEntity<PersonUpdateDTO> updatePersonById(@PathVariable Long id,

        @RequestBody @Valid PersonUpdateDTO personUpdatedDTO) {

        PersonUpdateDTO personResource = personFacade.updatePerson(id, personUpdatedDTO);

        return ResponseEntity.ok(person);

    }

}
```



# REST - Spring REST Controller

- CREATE:

```
@RestController
@RequestMapping(value = "/persons")

public class PersonRestController {

    @PostMapping(consumes = MediaType.APPLICATION_JSON_VALUE, produces = MediaType.APPLICATION_JSON_VALUE)
    public ResponseEntity<PersonCreatedDTO> createPerson(
        @Valid @RequestBody PersonCreateDTO personCreatedDTO) {
        PersonCreateDTO person = personFacade.create(personCreatedDTO);
        return ResponseEntity.ok(person);
    }
}
```



## REST - @RestController

- In the previous slides you may encounter that we use @RestController instead of @Controller from Spring-MVC
- @RestController is a stereotype annotation that combines @ResponseBody and @Controller
- More than that, it gives more meaning to your Controller and also may carry additional semantics in future releases of the framework
- In general, please use @RestController in your



# REST - HTTP Response Using ResponseEntity

- **ResponseEntity:** is meant to represent the entire HTTP response
  - You can control anything that goes into it:
    - Status code
    - Headers
    - Body
  - Basically, ResponseEntity lets you do more
  - E.g., returning null or void results in a 204 (No Content) status.. But in some cases you want to declare it as HTTP 200 OK

# REST - Serialization Configuration - Jackson

- The most common library for serialization in REST services is Jackson (also the default in Spring Boot)
- This configuration may include serialization of Dates, serialization of format (camelCase, snake\_case, ...)

@Configuration

```
public class ObjectMapperConfiguration {
```

@Bean

```
public ObjectMapper objectMapper() {
```

```
    ObjectMapper objectMapper = new ObjectMapper();
```

```
    objectMapper.setPropertyNamingStrategy(PropertyNamingStrategy.SNAKE_CASE);
```

```
    objectMapper.registerModule(new JavaTimeModule());
```

```
    objectMapper.disable(SerializationFeature.WRITE_DATES_AS_TIMESTAMPS);
```

```
    objectMapper.enable(SerializationFeature.INDENT_OUTPUT);
```

```
    return objectMapper;
```

```
}
```

```
}
```

Java POJO with camelCase letters  
will be serialized into snake\_case and  
vice versa

Dates will be serialized  
into one field

Will be serialized with  
indented output



# REST - Exception Handling

- Any unhandled exception will cause an HTTP 500 response (INTERNAL SERVER ERROR)
- To avoid returning HTTP 500 response for all exceptions you could annotate exceptions with `@ResponseStatus` to return appropriate HTTP error code and message

```
@ResponseStatus(value = HttpStatus.NOT_FOUND, reason = "The requested resource was not found")
```

```
public class ResourceNotFoundException extends RuntimeException {...}
```

- Methods annotated with `@ExceptionHandler` are handling exceptions
- It is not necessary to add `@ResponseStatus` to the Exceptions, it gives you more freedom in returning a custom error data structure

```
@ExceptionHandler @ResponseStatus(value = HttpStatus.NOT_FOUND, reason = "The requested resource was not found")
```

```
ApiErrorTraining handleResourceNotFound(ResourceNotFoundException ex) {  
    ApiErrorTraining apiError = new ApiErrorTraining();  
    apiError.setErrors(Arrays.asList("The requested resource was not found"));  
    return apiError;}  
}
```



# REST - Global Exception Handler

- Another way is to have a global advice using `@RestControllerAdvice` that will manage exceptions for all the controllers
- Personally, I like that way

```
@RestControllerAdvice
```

```
public class CustomRestExceptionHandlerTraining extends ResponseEntityExceptionHandler {  
    ...  
    @ExceptionHandler({Exception.class})  
    public ResponseEntity<Object> handleAll(final Exception ex, final WebRequest request, HttpServletRequest req) {  
        final ApiErrorTraining apiError = new ApiErrorTraining  
            .ApiErrorBuilder(HttpStatus.INTERNAL_SERVER_ERROR,  
                ex.getLocalizedMessage()).setError("error").setPath(URLHELPER.getRequestUri(req)).build();  
        return new ResponseEntity<>(apiError, new HttpHeaders(), apiError.getStatus());  
    }  
}
```



# REST - Swagger Documentation

- Moreover, every change in the API should be simultaneously described in the reference documentatie. Accomplishing this manually is a tedious exercise, so automation of the process was inevitable -> Swagger
- Swagger is annotation-based documentation
- Add Maven dependency:

```
<dependency>  
  <groupId>io.springfox</groupId>  
  <artifactId>springfox-swagger2</artifactId>  
  <version>${swagger.version}</version>  
</dependency>
```





# REST - Swagger Spring Configuration

@Configuration

@EnableSwagger2



Enabled through this annotation

```
public class SwaggerConfig {
```

```
    @Bean
```

```
    public Docket api() {
```

```
        return new Docket(DocumentationType.SWAGGER_2)
```

```
            .groupName("public-api")
```

```
            .apiInfo(apiInfo()).useDefaultResponseMessages(false)
```

```
            .select()
```

```
            .apis(RequestHandlerSelectors.any())
```

```
            .paths(PathSelectors.any())
```

```
            .build();
```

```
    }
```



## REST - Verification That It Works

- To verify that Springfox is working, you can visit the following URL in your browser

<http://localhost:8080/sedaq-rest/api/v2/api-docs>

- The result is a JSON response with a large number of key-value pairs, which is not very human-readable
- Fortunately, Swagger provides **Swagger UI** for this purpose



# REST - Swagger UI


- Swagger UI is a built-in solution which makes user interaction with the Swagger-generated API documentation much easier
- To use Swagger UI, one additional Maven dependency is required:

```
<dependency>
  <groupId>io.springfox</groupId>
  <artifactId>springfox-swagger-ui</artifactId>
  <version>${swagger.version}</version>
</dependency>
```

- Now you can test it in your browser by visiting *<http://localhost:8080/your-app-root/swagger-ui.html>*



# REST - Swagger UI

 **swagger**

Select a spec: public-api

## REST API documentation

[ Base URL: localhost:8080/sedaq-rest-training/api/v1 ]  
<http://localhost:8080/sedaq-rest-training/api/v1/v2/api-docs?group=public-api>

Developed By Pavel Seda  
[Terms of service](#)

basic-error-controller

Basic Error Controller

>

meeting-rest-controller

Meeting Rest Controller

>

operation-handler

Operation Handler

>

person-rest-controller

Person Rest Controller

>

web-mvc-links-handler

Web Mvc Links Handler

>

Models

>



# REST - Swagger UI

GET

/persons/{id} Get Person by Id.

Cancel

Parameters

Name	Description
Person Id <small>integer(\$int64)</small> <small>(path)</small>	id <div>Person Id - id</div>
fields <small>string</small> <small>(query)</small>	Fields which should be returned in REST API response <div>fields - Fields which should be returned in RE</div>

Execute

Responses

Response content type application/json

Code	Description
200	<div>OK</div> <div>Example Value   Model</div> <div><pre>{   "age": 98,   "birthday": "2018-11-21",   "email": "pavel.seda@gmail.cz",   "first_name": "Pavel",   "id_person": 1,   "nickname": "SedaQ",   "surname": "Seda" }</pre></div>
404	<div>The requested resource was not found.</div>



# REST - Configuration of Controllers for Swagger

- At class level use @Api annotation

```
@Api(value = "/persons", consumes = MediaType.APPLICATION_JSON_VALUE,  
      produces = MediaType.APPLICATION_JSON_VALUE)  
  
@RestController  
  
@RequestMapping(value = "/persons")  
  
public class PersonRestController {  
  
    ...  
  
}
```



# REST - Configuration of Controllers for Swagger

- At method level use @ApiOperation, @ApiResponses, @ApiParam annotations

```
@ApiOperation(httpMethod = "GET", value = "Get Person by Id.",
    response = PersonDTO.class, nickname = "findPersonById",
    produces = MediaType.APPLICATION_JSON_VALUE)

@ApiResponses(value = {@ApiResponse(code = 404, message = "The requested resource was not found.")})

@GetMapping(path =("/{id}", produces = {MediaType.APPLICATION_JSON_VALUE})

public ResponseEntity<Object> findPersonById(

    @ApiParam(name = "Person Id") @PathVariable Long id) {

    ...

}
```



# REST - Configuration of DTOs for Swagger

- At DTO classes use `@ApiModel` at class level and `@ApiModelProperty` at fields level

```
@ApiModel(value = "PersonDTO", description = "Information about person.")

public class PersonDTO {

    @ApiModelProperty(value = "Person ID.", example = "1")

    private Long idPerson;

    @ApiModelProperty(value = "Person email.", example = "pavelseda@email.cz")

    private String email;

}
```





# REST - Swagger in PDF?

- Could be configured in Maven plugins using:
  - swagger-maven-plugin
  - swagger2markup-maven-plugin
  - asciidoctor-maven-plugin
- For more information see:
  - <https://github.com/SedaQ/rest-training/blob/master/rest/pom.xml>

## 1. Overview

API Reference Description.

### 1.1. Version information

Version : v1

### 1.2. Contact information

Contact : Pavel Seda

Contact Email : [pavelseda@email.cz](mailto:pavelseda@email.cz)

### 1.3. URI scheme

Host : localhost:8080

BasePath : /rest-training/api/v1

Schemes : HTTP, HTTPS

### 1.4. Tags

- meetings
- persons

## 2. Paths

### 2.1. Get All Meetings.

GET /meetings

#### 2.1.1. Parameters

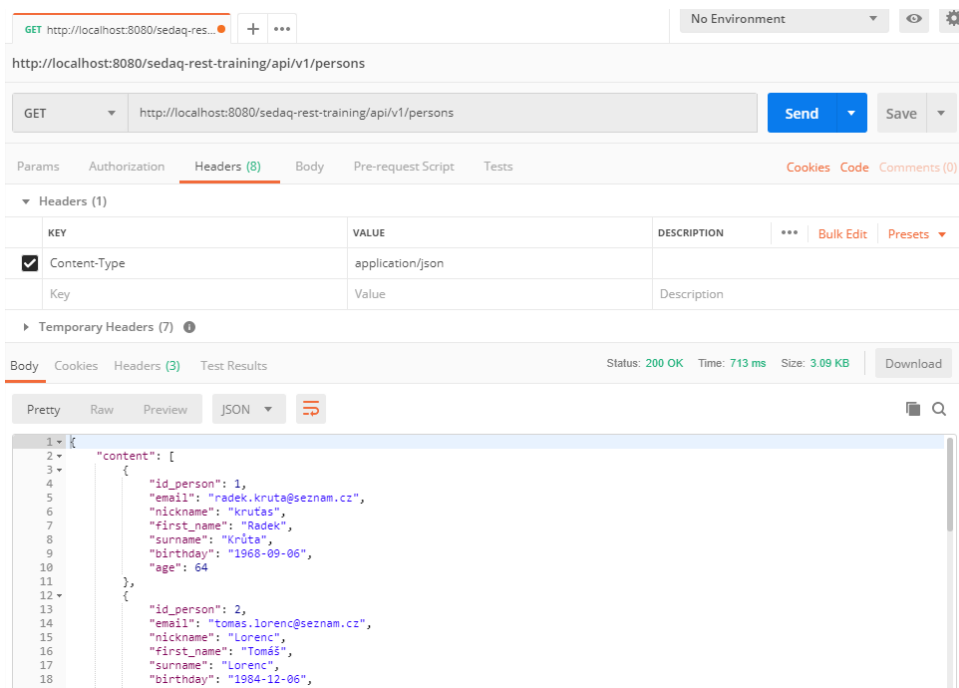
Type	Name	Description	Schema	Default
Query	<b>fields</b> <i>optional</i>	Fields which should be returned in REST API response	string	
Body	<b>body</b> <i>optional</i>	Parameters for QueryDSL filtering	< string, < string > array > map	
Body	<b>body</b> <i>optional</i>		< string, < string > array > map	



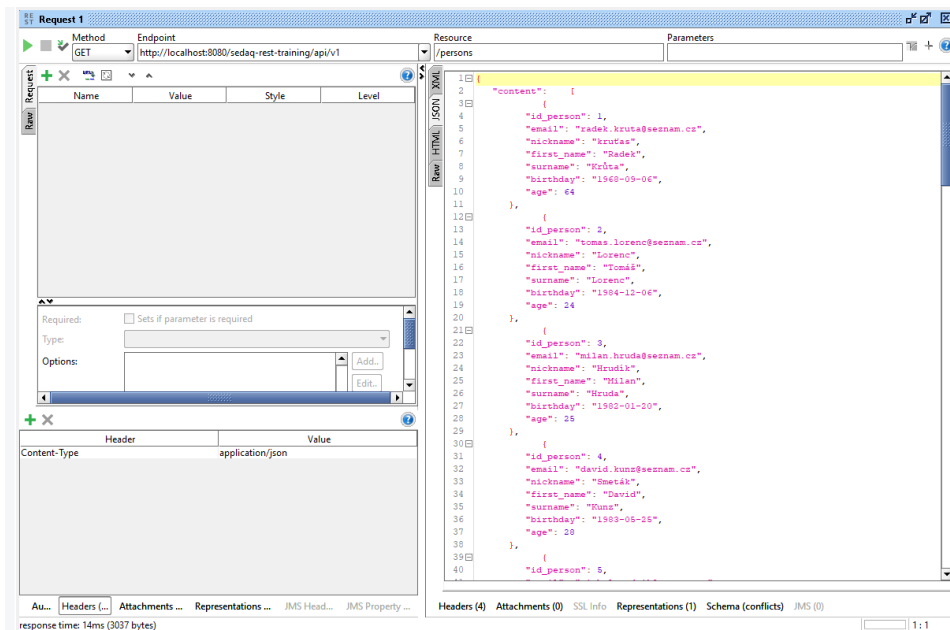
# REST - Testing Tools

- Postman
  - The most common for testing REST services
  - User friendly but contains less features as SoapUI
- SoapUI
  - The most common for testing SOAP services
  - Contains a lot of features, but less user friendly as Postman
- Browser plugins
  - **Chrome:** Restlet Client <https://chrome.google.com/webstore/detail/restlet-client-rest-api-t/aejoelaoggembcahagimdiliamlcdmfm>
  - **Firefox:** Advanced Rest Client <https://chrome.google.com/webstore/detail/advanced-rest-client/hgmloofddfdnphfgcellkdfbfbjeloo>

# REST - Testing Tools - Postman



# REST - Testing Tools - SoapUI





## **Section: Spring Security**



# Spring Security

- a. Configuration of Spring Security
- b. Securing web requests
- c. Storing password in Database



# Spring Security

- Authentication
  - Is the process of determining whether someone or something is, in fact, who or what it declares itself to be
  - Verified by, e.g., HTTP Basic form (username, password combination)
- Authorization
  - Determining if authenticated entity has permitted access to a protected resource or system
- Confidentiality
  - Is a set of rules or a promise that limits access or places restrictions on certain types of information
- Data integrity
  - Received data has not been modified, destroyed or lost

Solved by  
SSL





# Login Type

- We must determine what type of authentication we will be using
- There exists 4 main types of authentication:
  - **BASIC:** Transmits data unencrypted
  - **FORM:** Similar to BASIC but allows user to add input data (username, password) to the formular
  - **DIGEST:** Sends data encrypted
  - **CLIENT-CERT:** Authentication through client certificate. It is quite safe type of authentication, but the client must have a certificate on his device

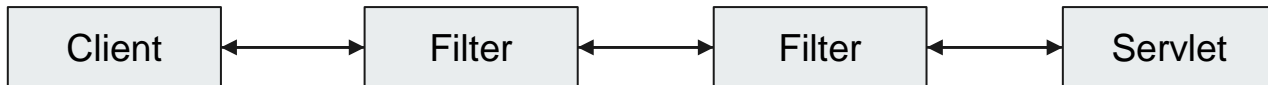


# Spring Boot Security

- Add the following maven dependency:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <scope>test</scope>
</dependency>
```

- Spring Security in the web tier (for UIs and HTTP back ends) is based on Servlet Filters, so it is helpful to look at the role of Filters generally first (<https://spring.io/guides/topicals/spring-security-architecture>)





# Spring Boot Security

- The client sends a request to the app, and the container decides which filters and which servlet apply to it based on the path of the request URI
- At most one servlet can handle a single request, but filters form a chain, so they are ordered, and in fact a filter can veto the rest of the chain if it wants to handle the request itself
- A filter can also modify the request and/or the response used in the downstream filters and servlet



# Spring Boot - @EnableWebSecurity

- To configure Spring Security create config class with @EnableWebSecurity annotation

```
@Configuration
@ComponentScan("com.sedaq.training.security")
@EnableWebSecurity

public class RestSecurityConfiguration extends WebSecurityConfigurerAdapter {

    @Override

    protected void configure(AuthenticationManagerBuilder auth) throws Exception {

        auth.authenticationProvider(authProvider);

    }

    ...

}
```



# Spring Boot - Specify Authentication Rules

- Configure `HttpSecurity` for certain URLs or Resources

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    Http
        .authorizeRequests()
        .antMatchers("/v2/api-docs", "/swagger-resources", "/swagger-ui.html", "/webjars/**")
        .permitAll()
        .and()
        .authorizeRequests()
        .antMatchers("/**")
        .authenticated()
        .and()
        .httpBasic();}
```



# Spring - Authorization Checks

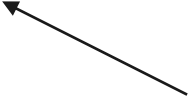
- Spring provides several annotations to check if a user has given roles:
  - `@Secured`, e.g., `@Secured("ROLE_USER")`
    - The `@Secured` annotation doesn't support Spring Expression Language (SpEL)
  - `@RolesAllowed`, e.g., `@RolesAllowed("ROLE_USER")`
    - The `@RoleAllowed` annotation is the JSR-250's equivalent annotation of the `@Secured` annotation
    - Similar to `@Secured`, does not support SpEL
  - `@PreAuthorize`, e.g., `@PreAuthorize("hasRole('ROLE_VIEWER')")`
    - Supports SpEL, the strongest variant, I prefer this one



# Spring - Authorization Checks - Own Annotation

- It is common to create own stereotypes for security checks, e.g., for user it would look like:

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@PreAuthorize("hasRole('USER')")
@Documented
@Inherited
public @interface IsUser {
}
```



Defines that it must be  
authenticated user with role  
USER



# Enforce HTTPs

- We must determine what type of authentication we will be using
- There exists 4 main types of authentication:
  - **BASIC:** Transmits data unencrypted
  - **FORM:** Similar to BASIC but allows user to add input data (username, password) to the formular
  - **DIGEST:** Sends data encrypted
  - **CLIENT-CERT:** Authentication through client certificate. It is quite safe type of authentication, but the client must have a certificate on his device



## Summary

- After this part you should be able to:
  - Create a simple Spring Boot application
  - Design n-tier application
  - Design and implement JPA entities and repositories
  - Design and implement business logic for Spring-based applications
  - Design and implement REST architecture using Spring MVC
  - Understand and use Spring Security